MODELLING GUIDELINES FOR THE COMVANTAGE ACADEMIC PROTOTYPE ON OMILAB[1]

Authors: Dimitris Karagiannis, Robert Buchmann, Patrik Burzynski

Contents

[1] Adapted from public deliverable D3.5.2 available at http://www.comvantage.eu/results-publications/public-deriverables/

1 Access Means to Modelling Prototype

The academic prototype ("the OMI modelling prototype") has been deployed on the Open Model Initiative portal for registered members[2], with RDP on-line access (means of access are provided upon registration on OMI). It provides a partial coverage of the ComVantage method's metamodel for academic experimentation purposes.

Registration on the OMILab community space enables access to all resources pertaining to this prototype. OMILab registration must be followed by membership request to the ComVantage tool, as highlighted in Figure 1, to obtain tool credentials and rights on shared model folders.

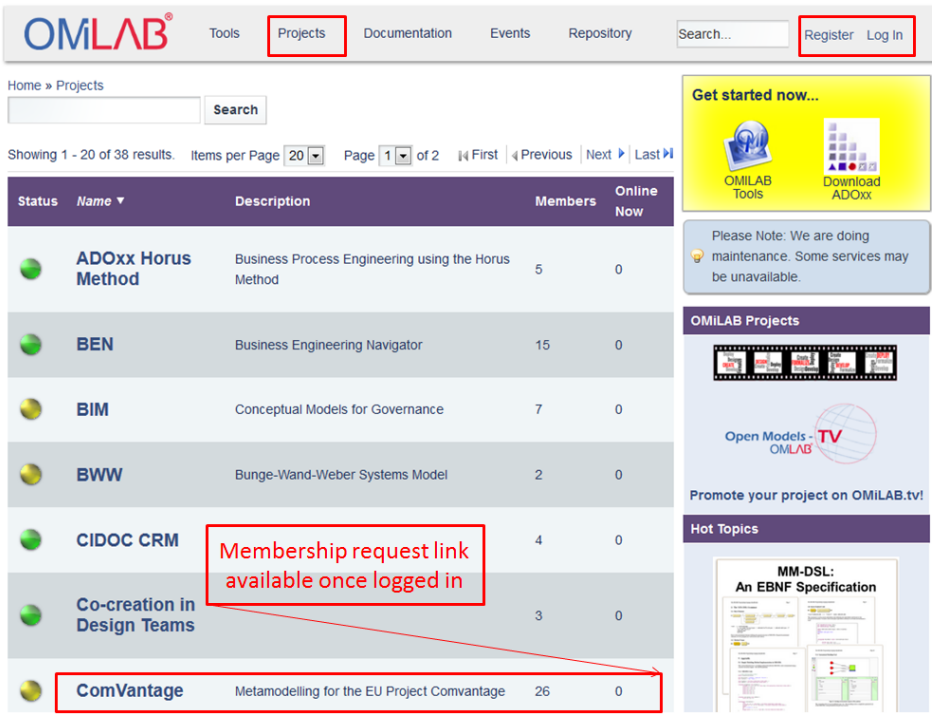

Figure 1: The OMILab portal

2 General User Interface

The prototypical implementation makes use of the metamodelling platform hosted by OMI. This platform provides some general functions independent of the modelling method. An overview of the platform's user interface template (extended for our implementation) is shown in Figure 2.

The platform comes with many different functions, most of which can be accessed in different ways. For example to delete an object either select it and press the "Del" key or right click on it to open the context menu and select "Delete". For the guidelines it is assumed that the user already has some knowledge of using software in general (e.g. how to save, what the "Open" menu does, how to copy and paste, drag and drop, tooltips etc.) as well as some basic knowledge about conceptual modelling. Therefore, these guidelines will mostly present one way to achieve something specific. The user is invited to experiment with the platform and prototype on their own. Additional help can be found in the "Help" menu of the platform.

[2] http://www.omilab.org/web/comvantage/home

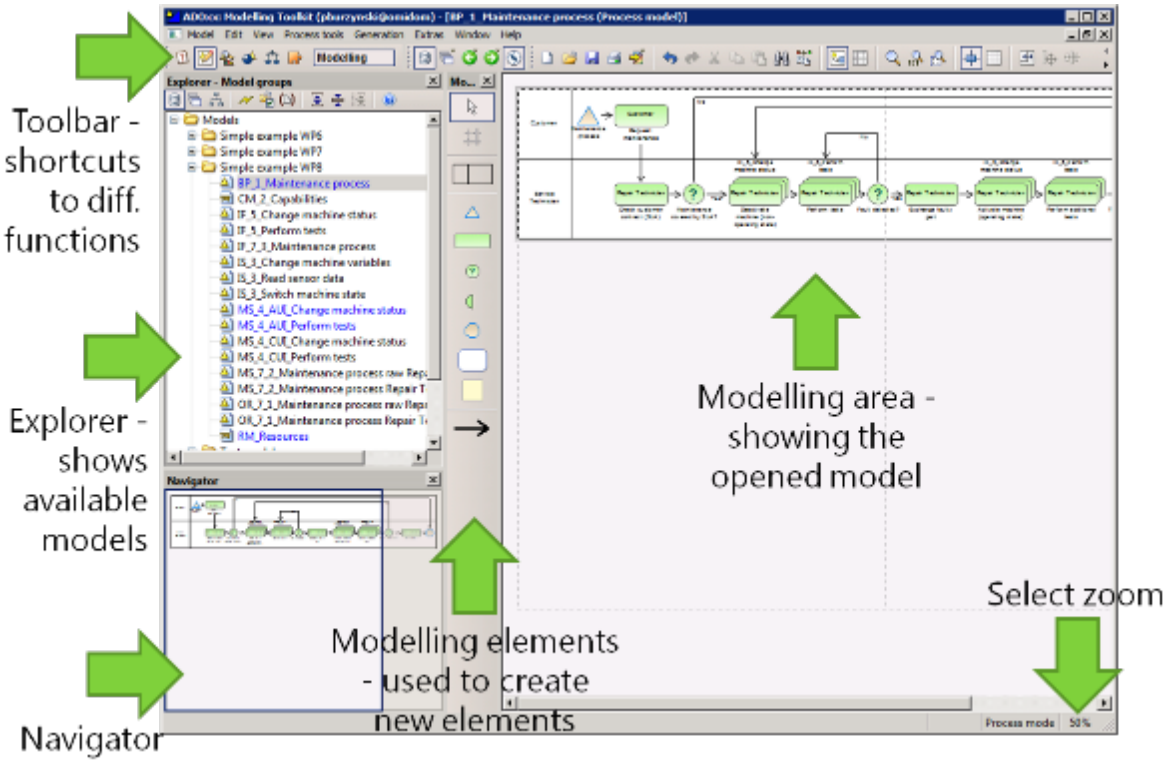

Figure 2: Prototypes general user interface

Certain elements like the *Explorer* and the *Navigator* can be closed to make more room for the *Modelling area*. If they are missing, they can be reactivated through the "Tools" submenu in the "Window" menu at the top. The user interface also changes depending on the selected component. Changing the component mostly influences the available menus and toolbar icons. The components can be switched using the corresponding icons in the toolbar shown in Figure 3.

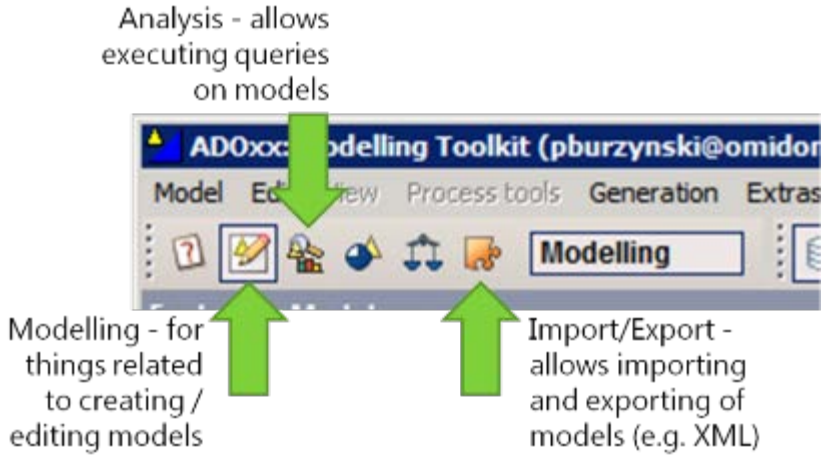

Figure 3: Available components

The relevant components in this prototype are *Modelling*, *Analysis* and *Import/Export*. They are described in further detail in the following sections.

3 Modelling

The *Modelling* component deals with the creation, modification and management of models inside the platform. For this the *Explorer* is most helpful since it provides a list of all models available to the user. The models themselves are organized in a tree of model-groups. The structure can be compared to the general file system where files (models) are inside folders (model-groups).

To create a new model right click in the *Explorer* on the model group where it should be created and select the desired model type in the sub menu "New" of the context menu. This can be seen in Figure 4. Model-groups are created in a similar fashion.

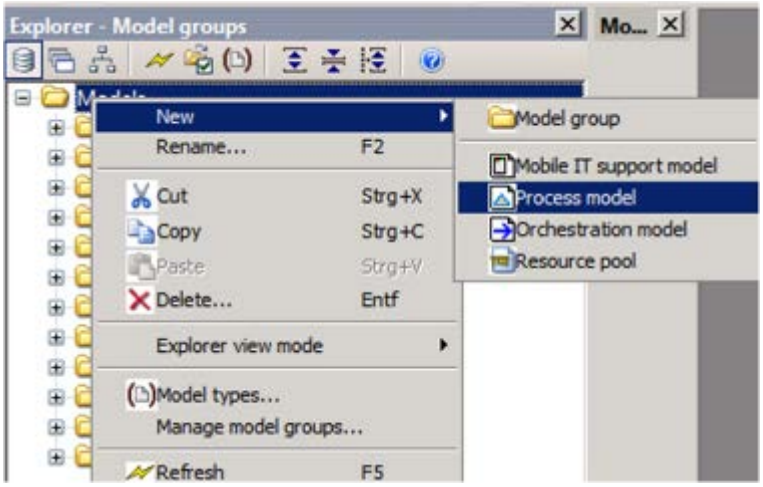

Figure 4: Create a new Process model

To rename a model right click in the *Explorer* on it and select "Rename" in the context menu. This will open a dialog asking for the name and version for the model. The version is shown as a suffix to the model name and both merged can be considered the full name of the model. At no time can there be two models of the same type with the same full name, independent of the model-group they are in.

An additional functionality has been added to the OMI modelling prototype in order to **clone models**. This is necessary since several partners are working on the models with different read and write accesses[3] and sometimes they need to copy models in order to edit them in their own folders rather than working on the original models. Through the cloning a set of models can be copied with proper relations between the clones (according to the original models), which is not the case when copy and pasting the models. The functionality can be accessed in the "Cloning" menu through the "Clone" item. This will show a popup where the models to be cloned have to be selected. After clicking on OK a new popup will ask in which model-group the clones should be created. Selecting OK will show a third popup asking for a suffix which will be attached to the name. This is necessary since the full name of a model has to be unique. After all this information has been provided the clones will be generated in the target model-group.

To create objects in a model first open it by double clicking on it in the *Explorer*. This will open and show the model in the *Modelling area* and the available concepts for that model type in the *Modelling elements* bar[4]. Once the model is opened simply select the desired concept (e.g. *Process start*, *Activity* etc.) from the *Modelling elements* bar and click in the *Modelling area* where the object should be placed (see Figure 5).

[3] Those access rights have been set to prevent one partner from changing someone else's models.

[4] The list of available concepts is also influenced by the selected mode, which can be changed through the "Mode" submenu of the "View" menu.

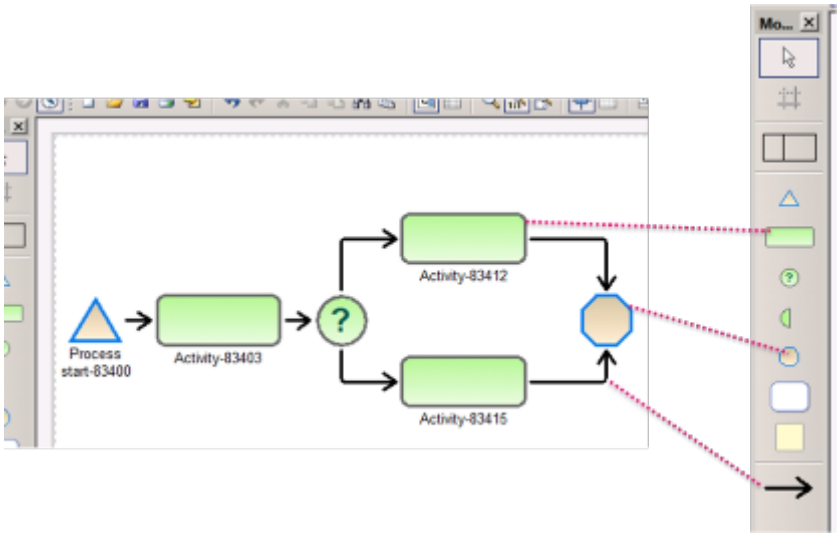

Figure 5: Modelling elements bar for process model type

Relations (e.g. *Sequence relation*) on the other hand are drawn between two objects, a source and a target object. To create a relation first select it in the *Modelling elements* bar, then select the source of the relation and then the target. Additional points called bend points can be added to put angles in the relation. To do this select the relation, click on its line and drag it to a new place in the *Modelling area*. The bend point is represented by a white rectangle when the relation is selected. To remove it, simply drag it over an already existing bend point. In addition to the bend points two other things are visualized when a relation is selected. The position of text displayed by the relation is represented by a green diamond and the yellow circles allow changing the source and target objects. An example relation with six bend points and "Some text" attached to it between two objects can be seen in Figure 6. In case a relation does not have a notation or is between objects in different models it is implemented as a reference. This means it can be found as an attribute of the source object.

Figure 6: A relation with bend points

The model itself has a certain size which can be changed. A new model has by default the size of an A4 sheet. Also the size of the model can never be smaller than the space that is required by its contents. To change the size of a model, usually making it smaller, click and drag the bottom right corner of the canvas as seen in Figure 7. Note that this changes the size of the model, not the zoom factor. To change the zoom, select "Zoom" in the "View" menu or double click on the percentage value at the bottom right of the main window.

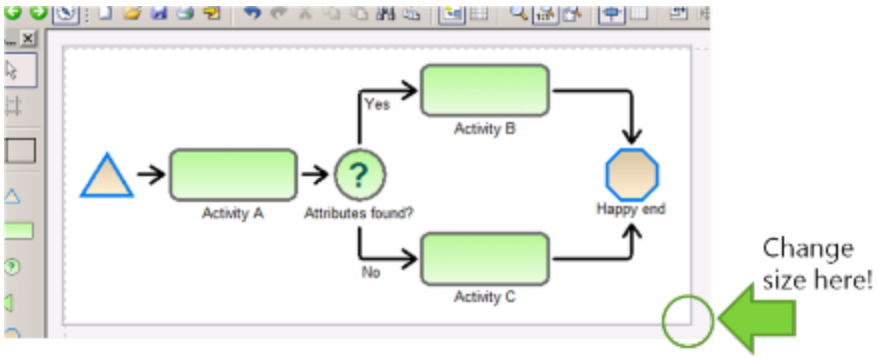

Figure 7: Change model size

In order to change the attributes of an object, double click on it in the *Modelling area* which opens the *Notebook* containing its attributes with values. Models themselves can also have attributes. The *Notebook* containing the model attributes can be accessed through "Model attributes" in the "Model" menu. An example for a *Notebook* can be seen in Figure 8. In the *Notebook* the attributes are organized in chapters. The arrows at the bottom of the *Notebook* can be used to change pages in case a chapter contains more attributes than can be shown. The type of the attribute controls how it is shown in the *Notebook* and how it can be edited. In this platform the type of the object, its name and the model to which it belongs are used to identify it. This means that one model cannot contain two objects of the same type with the same name[5].

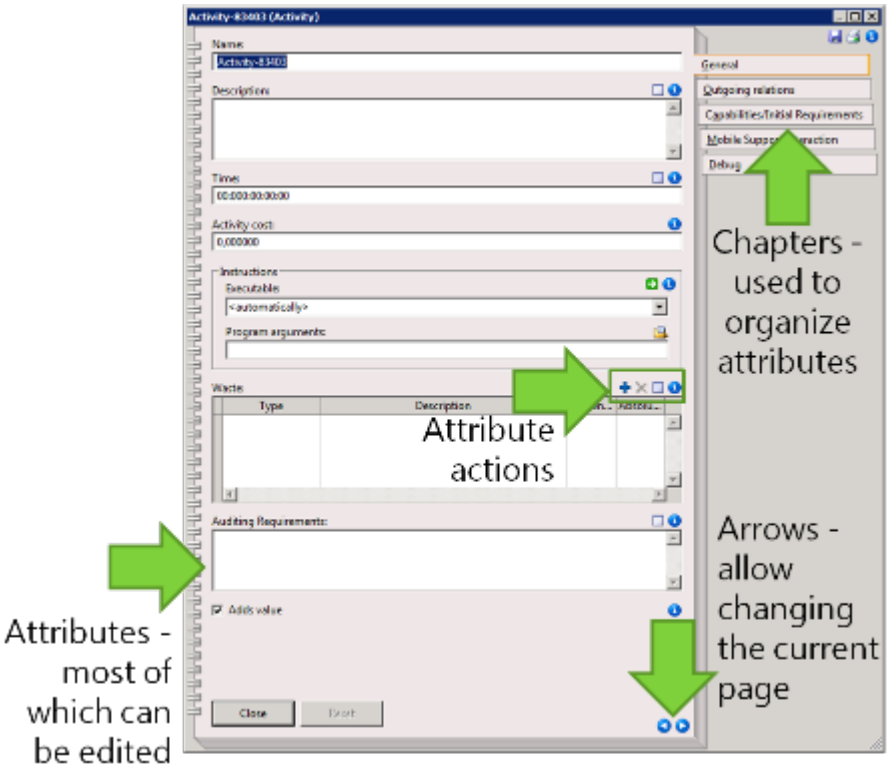

Figure 8: Example for a *Notebook*

Attribute actions like the ones seen in Figure 9 can appear above an attribute field. Which ones are shown depends on the attribute type and available information. Their meaning from left to right is:

1. Add – Adds a row to a table or an inter-model relation (relation between different models).
2. Delete – Removes the selected row or inter-model relation.

[5] If it is absolutely vital that two objects pretend to have the same name then whitespaces at the end of the name can be used as a work around.

3. Execute / Follow
    a. Execute passes certain attributes to the operating system.
    b. Follow jumps to the target of an inter-model relation.
4. Browse – Allows selecting a file through the file browser of the operating system.
5. Dialog – A special dialog which supports editing the attribute value.
6. Info text – Provides an information text about the attribute.

Figure 9: Attribute actions

It is important to remember that relations which connect objects between two different models or which don't have a notation (here called references) are also found in the *Notebook* of the source object together with the attributes. To see if and which such relations point towards a certain object, or in other words find out for which relations the object is the target of, simply right click on the object ("the target", for example an instance of *Role*) and select "Object references" in the context menu as shown in Figure 10[6]. This will show a dialog with all the objects which are in a relation with "the target" and from which model they are. They are grouped by the relation type in which they participate (e.g. *Assigned role*) and their object type (e.g. *Activity*). Selecting an object in this dialog and clicking on "Follow" will jump directly to that object and of course open the corresponding model.

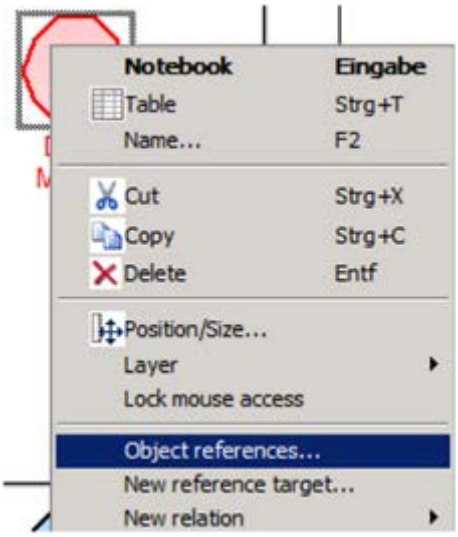

Figure 10: Traverse a relation without a notation in the opposite direction

An additional functionality for handling references has been added to the OMI prototype called **Multi-linking**. It allows adding or replacing several references for multiple objects at once based on a queue. To make use of it, first select the objects, which should be the source of the reference (i.e. in whose notebook the reference should be set) and add them to the queue using the "Add to Queue" item in the "Multi-linking" menu. This can be performed several times and the objects can be of different classes. The queue can also be cleared using the "Clear Queue" menu item. Once the desired objects are in the queue go to the model containing one or more of the elements that should be the target of the reference. If a model should be the target simply open the model and ensure that nothing is selected. Then select the "Process Queue" menu item. This opens a window that shows a list of possible reference attributes (taken from all of the objects in the queue based on their class). Select the attribute that should be used, set the checkboxes at the bottom accordingly (use "Remove links before adding." to set the attribute instead of adding to the attribute) and select OK.

[6] To do this for a model just right click on an empty space in the *Modelling area* and select "Model references".

This will add (or set) the selected objects as the target of the reference for all objects of the queue in the specified attribute where possible. If an object of the queue does not contain/allow the attribute then it will be skipped.

The Modelling component also provides functionality to create a graphic out of the selected model. It can be accessed through the menu item “Region” which can be found in “Generate graphics” in the “Edit” menu. This will show a dialog where further details can be specified (e.g. file type, storage location, zoom etc.). It will create a graphic for the selected region in the currently open model, which by default is the entire drawing area. The region can be set by holding down the ALT key on the keyboard, performing a left click with the mouse in the *Modelling area* where one of the corners should be and dragging towards the opposite edge to create a region. The current region is represented by a blue rectangle which can be resized like a window and removed by clicking anywhere on the *Modelling area*.

4 Analyzing

The *Analysis* component allows executing queries using the proprietary language of the platform. However, the platform also provides a dialog which can be used to build simple queries. Two Examples for simple queries executed using the *Analysis* component are “What Activities have a cost higher than X?” or “Which Roles have no description specified?” Generally speaking the first example allows finding potential for optimization while the second example supports identifying incomplete objects.

To access the query dialog first select “Queries/Reports” in the “Analysis” menu[7] as shown in Figure 11. In the following dialog specify on which models the query should be executed. This will open the query building dialog as shown Figure 12. The top part provides some forms for query building blocks. By pressing the “Add” button the building block will be added to the actual query using the proprietary syntax in the bottom part of the dialog. The “and”/”or”/”diff” operators for the query can also be added there. Selecting “Execute” will run the current query on the previously selected models. An example result for a query asking for activities with costs of higher than 20 can be seen in Figure 13. For additional help use the corresponding button.

The tool can also have predefined queries, which run on specific types of models. They can be accessed through the same menu using the menu items in the “Queries on [Model type]” portion (seen Figure 11). This will open a model selection dialog, where the models should be selected. Afterwards a list of the available queries and a short description for each is shown and they can be executed on the previously selected models.

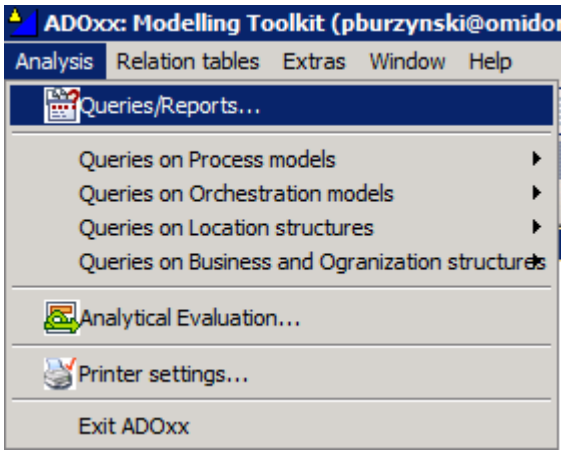

Figure 11: Open query dialog

[7] Make sure the *Analysis* component is selected in the tool bar if the menu isn’t there.

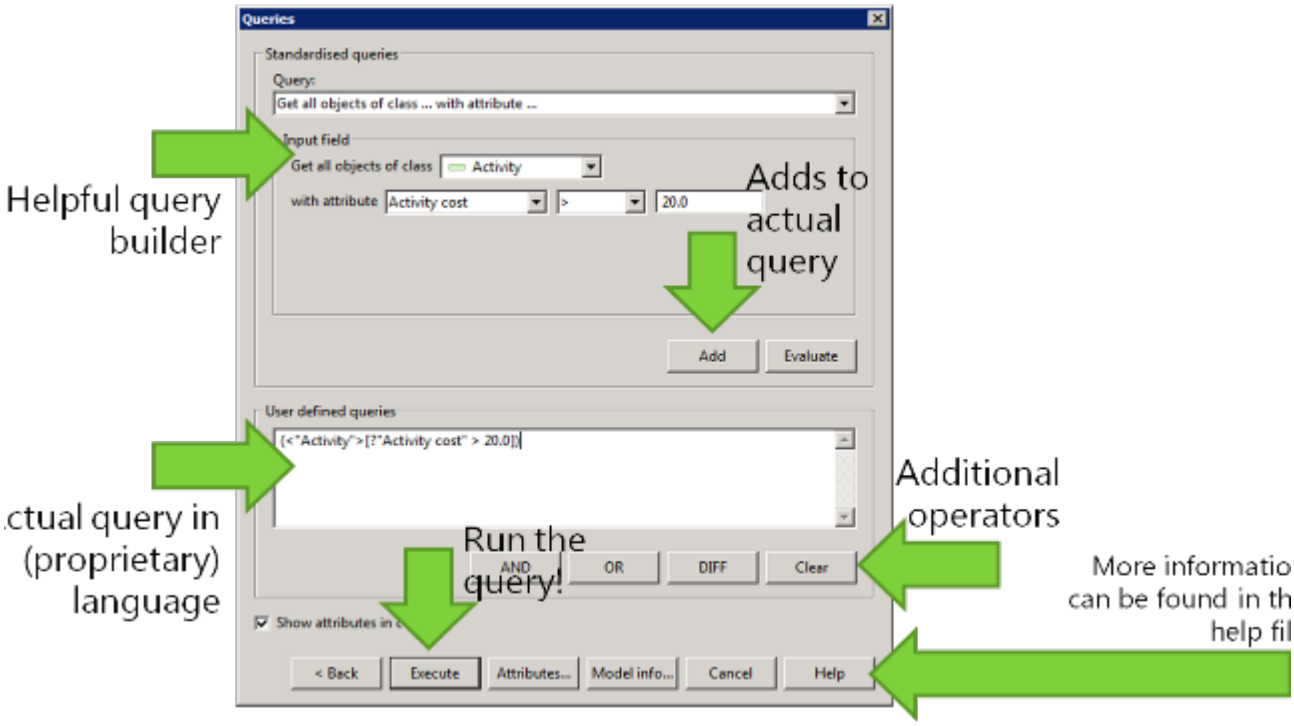

Figure 12: Dialog for building simple queries

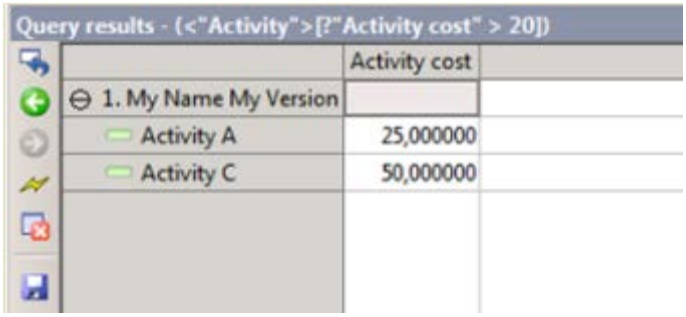

Figure 13: Example query result

5 Importing and Exporting

The *Import/Export* component allows storing models in files outside of the platform, where they can then be used as backups or for further processing. The models can be exported or imported from either the proprietary format ADL or using XML. The XML files of the export follow a Document Type Definition (DTD) which is also created when exporting one or several models. In those guidelines only the XML export and import is covered, since it is more versatile and usually enough for most cases.

To start the export first select "XML Export (default)" in the "Model" menu[8] as shown in Figure 14. This will open a dialog where the models and model-groups have to be selected for the export as seen in Figure 15. Take note of the checkboxes below the list for some additional export options. Some of those only become available when certain criteria are met (e.g. a model-group is selected). When using "Including referenced models" the reference depth has to be specified through the "References" button. Under "Export file" at the bottom of the dialog the location for the XML file to be created has to be provided. Clicking on "Browse" will open the file selection dialog as seen in Figure 16. However, be careful about the file selection. Since the prototype is running on a remote computer this also means that the "C:" drive is the local one of that computer. Instead, choose a drive in the "Other" category (e.g. "C on DELL 755-4" in Figure 16). After the location for the XML file

[8] Make sure the *Import/Export* component is selected in the tool bar if the menu isn't there.

has been specified a click on the “Export” button will start the export. When it is finished a result dialog will be shown. In addition to creating the file with the model data a file (“adoxml31.dtd”) containing the DTD for the XML structure is created in the same folder.

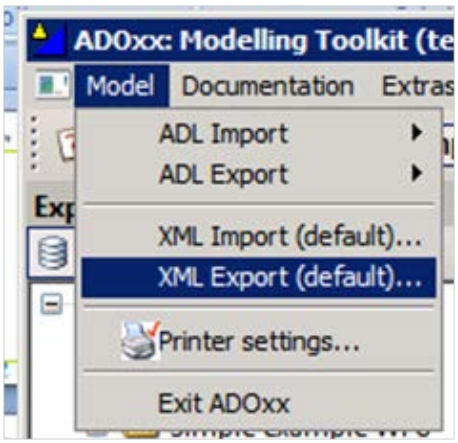

Figure 14: Start XML export

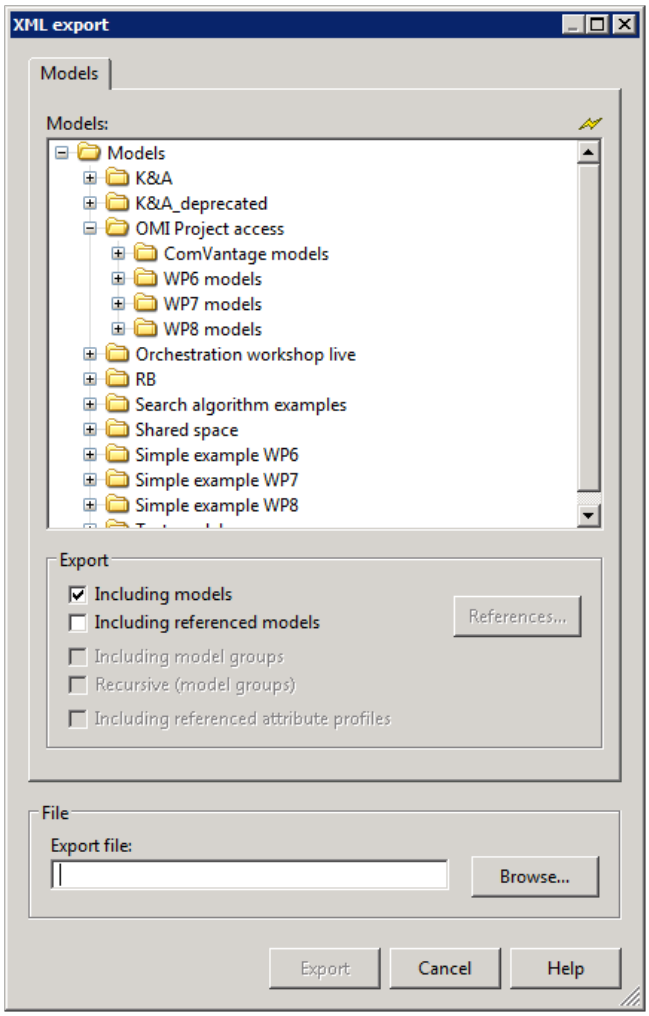

Figure 15: XML export model selection

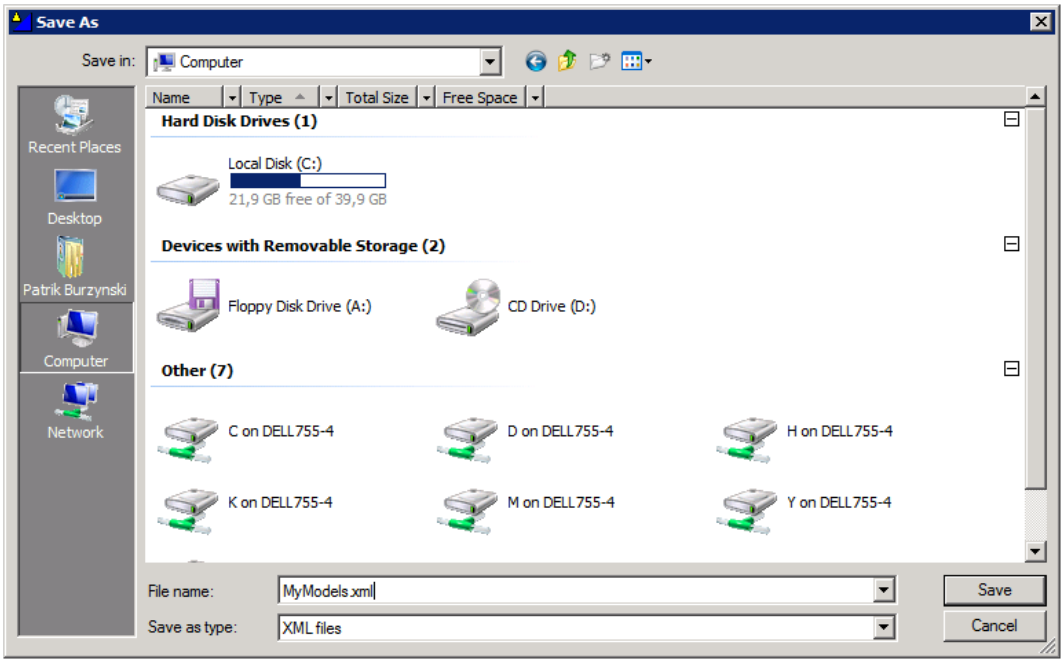

Figure 16: XML export file selection

To import models from an XML file select “XML Import (default)” in the “Model” menu, right above “XML Export (default)” in Figure 14. This will open the file selection dialog similar to Figure 16. Here select the XML file to import the models from. Again remember that the prototype is running on a remote computer and “C:” is that computers local drive. Note that the XML file has to follow the DTD provided by the platform, which is created when a model is exported. Also the DTD file has to be in the same folder as the XML file. Otherwise the XML parser will throw an error, because it cannot find the DTD. After selecting the file a dialog as shown in Figure 17 will ask where to import the models and model-groups. Again several options are available at the bottom of the dialog, like how to handle conflicts or to import the model-groups as well. After selecting “OK” the import will be performed and a result dialog will be shown.

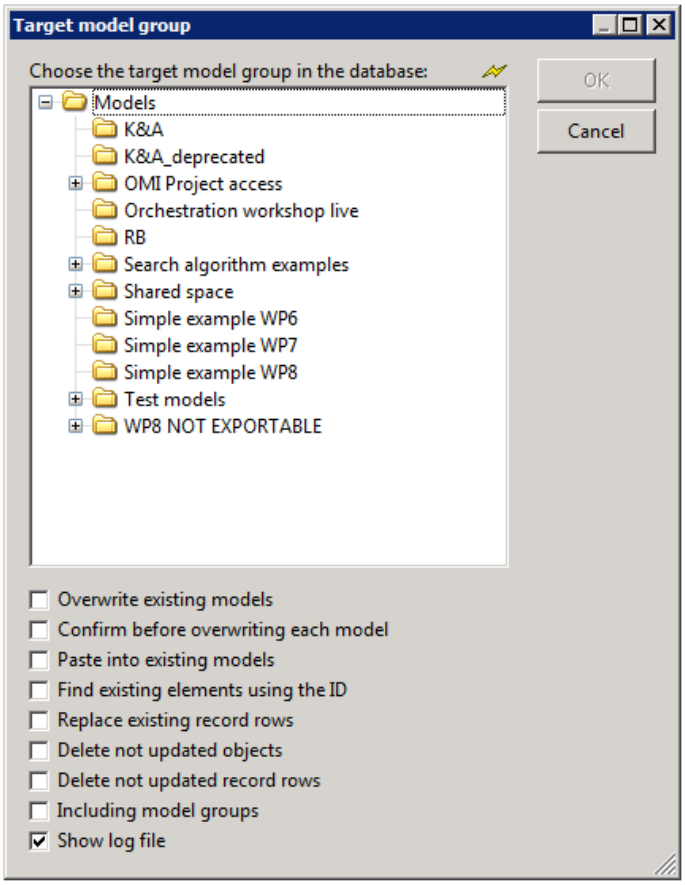

Figure 17: XML import model-group selection

6 ComVantage Modelling Guidelines

First, in section 1.1.2.1, some guidelines on how to create proper models will be given. Afterwards, in section 1.1.2.2, some procedures describing how those models can be used to achieve a certain goal, which can differ from procedure to procedure.

6.1 Model Type Guidelines

The current prototypical implementation makes use of ten model types. For each of those some guidelines will be provided on how to create them.

6.1.1 Model Type independent

One object type which is available in each model type is the *Note*. Through the *Note*, text for the human reader can be placed in a model. It can be used to describe assumptions that have been made or to point out problems or errors in a model which have to be solved or at least documented. The text is provided through its table attribute, which can have a date and an author attached to it. The *Note* will always show the last text entry in the table. Since it is drawn behind most objects it can also be used to seemingly group objects for the human reader. Additionally the *Attached to* relation can be used to attach notes to elements. Also a link to an image file can be provided which is then shown in the place of the *Note*. The supported file types are BPM, JPG, GIF and PNG among others. With all this it is however important to remember that the *Note* itself is not and should not be used beyond anything than to provide information to the human reader.

Additionally three attributes are available in many objects: the *sameAs link*, the *Property collector* and the *of type* attribute. The *sameAs link* should be used to specify a URI for an object if so desired. If it is empty then a URI should automatically be generated. The *Property collector* is a table that allows adding statements[9] for the exported RDF, where the element containing the table is

[9] In the typical RDF form of “Subject Predicate Object”

always the subject of the statement. The *of type* attribute is available in certain objects and allows to further type them using objects from the *Resource pool* (see section 1.1.2.1.2). For example the parts of a station can further be typed based on different *Component types* defined by the modeller.

Also as a general recommendation the flow of the main relations in models should adhere to the direction of reading. This means that models describing a sequence should have the objects arranged so the main flow goes from left to right and top to bottom. Models depicting a hierarchy on the other hand should have the root element at the top which is then decomposed towards the bottom.

6.1.2 Resource pool

The *Resource pool* model is used as a central repository to define custom types, mobile apps and their capabilities. Those objects are then reused through other models. Custom types are created by using one of the *Component type*, *Action type*, *Target type* or *Value type* classes. Other elements can then reference those types, usually through the *of type* attribute (e.g. *Activities* targeting *Action types*). A hierarchy can also be described between the types using the *is a* relation.

The *Mobile IT support features* can be compared to an app. They represent a feature that can be executed, support a user in some task and are meant to be run on a mobile device. The scope of a *Mobile IT support feature* is not fixed to a certain size and can therefore represent small and big apps or composed apps. Composed apps can use a reference to an *Orchestration model*, indicating from what other *Mobile IT support features* they are composed and how those are ordered. They can also be described through their input and output types, display properties, the platform they use, their capabilities and a Mobile IT support model (see section 1.1.2.1.6) with further details.

Capabilities represent abstract features or abilities that can be provided or required by a *Mobile IT support feature*. They are generally described through the types of components they require, the data they can work on and the type of action they consist of. Also a URI can be used to link them to further descriptions for example in the linked data cloud. Describing the capability requirements in a process and also providing the available capabilities of an app can be used together to find app candidates for the process.

The concepts in the *Resource pool* also has some overlapping with the App Description Language (ADL, see D5.3.3). This allows describing apps close to the structure of the ADL in the modelling tool, which can then be serialized as RDF, creating parts of the ADL description.

The left side of Figure 18 shows an example for a *Resource pool* with a *Mobile IT support feature*, the *Capability* it provides and several different *Target types* and *Action types*. A lot of the description is handled through attributes and not directly visualized as relations. The right side of the figure visualizes some of those relations for the model.

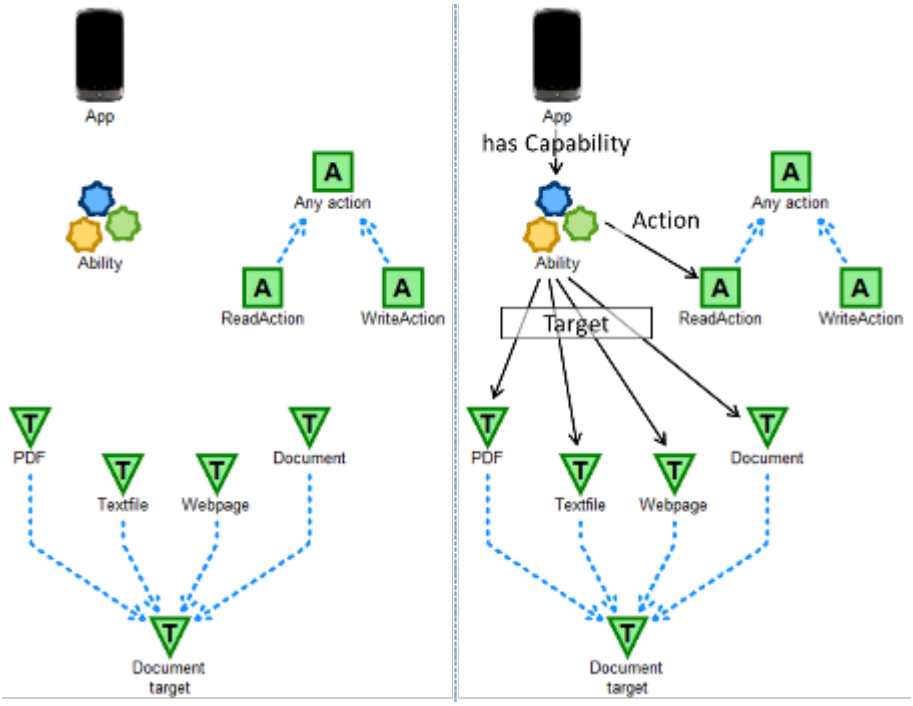

Figure 18: Resource pool example in the modelling tool (left) and with visualized references (right)

Other types of elements are also available in the *Resource pool* in order to preserve older models. All of those elements can be found in other model types and should be used there.

6.1.3 Business and Organization structure

The *Business and Organization structure* model is used to describe both the businesses that are of interest and their organizational structure. The businesses are described through *Business roles*, to depict types of businesses, and *Business entities*, actual instances of those types. For both it is possible to describe capabilities that they provide (e.g. the service of sewing a shirt for a certain price) and what resources they can/do own (e.g. an entity owns a sewing station). *Business entities* can be further described through their contact information, registry and tax number and can have a chief.

Business entities can be further decomposed into *Organization units* and/or *Performers,* which represent the humans working at a company. It is also possible to describe *Roles*. With a *Role* a certain task range can be described either in regard to function (e.g. “SAP Expert”) or in form of a position (e.g. “CEO”). *Roles* represent a certain skill-set that is either required by an *Activity* or covered by a human that can take on the role (indicated through the *fulfils* attribute). *Skills* and *Knowledge* elements can be further used to detail both *Roles* and *Performers* through their *Skills/Knowledge* table attribute. The *Skills/Knowledge* table attribute also allows to reference elements from other model types (e.g. *Machine types* to indicate knowledge about a certain type of machine).

Roles, *Business roles*, *Skills* and *Knowledge* can be arranged in a hierarchy through the *is a* relation. This relations is easiest to understand as substitution, where the use of the target of the *Is-a* relation could be replaced by the source of the *Is-a* relation.

Figure 19 shows an example for a *Business and Organization structure* with all of the available element types, except for *Knowledge*. The relation indicating the fulfilment of a role by an entity/performer is indicated through the red arrows in the elements, since it is handled through an attribute.

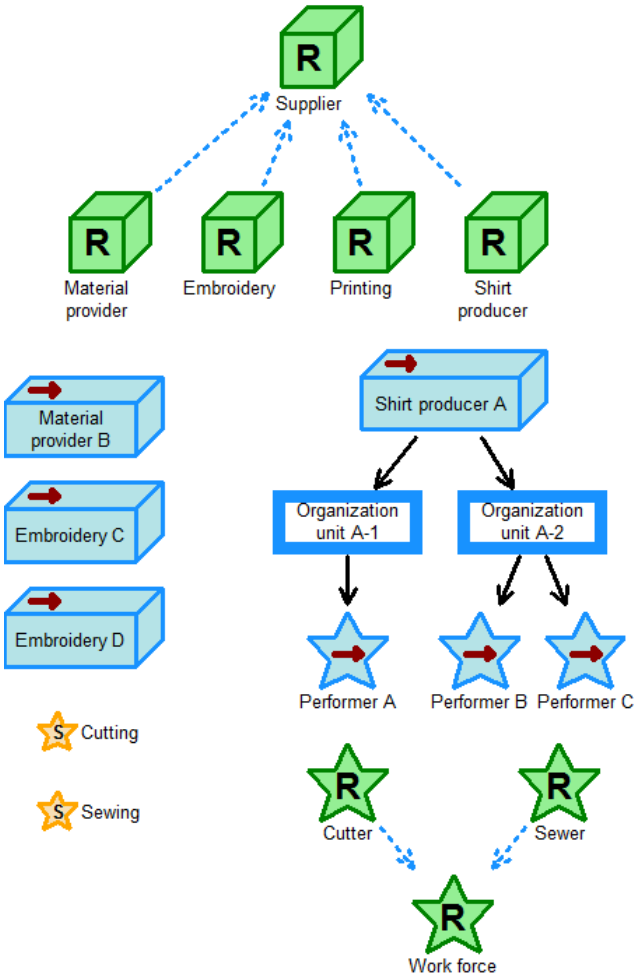

Figure 19: Business and Organization structure example

6.1.4 Value structure

A *Value structure* model describes the structure of different values (usually products) focusing on their decomposition. As such the core concept is the *Value*, which can be one of many different subtypes (e.g. Marketed or Not-marketed, it can be a Material, a Component, a Commodity, a Service or an abstract Value etc.). The major means of further describing a *Value* is by decomposing it through the use of the *has* relation into other *Values* or *Value sets* and their quantities. The latter concept can be used to describe variability and indicate where choices have to be made. The *separable* attribute of the *has* relation can be used to indicate optional parts, while the *Selectable by customer* allows to specify which of the choices should be decided by the customer.

The *Value* also has several relations to other values realised through attributes. The *implies* and *prohibits* attributes can be used to indicate the necessity or forced absence of other *Values*, should a certain value be chosen. Since *Values* can actually represent several "atomic values" (due to the possibility of choices), it is also possible to describe specializations using the *configuration of* attribute that represent a smaller set. The *opposite of* relation can be used to link *Values* (i.e. where larger quantities are generally better, e.g. money, available time) and their respective *Anti-Values* (i.e. where larger quantities are unwanted, e.g. costs, used up time), which are controlled through the *Axiological type* attribute. Also the party responsible for the provision of a certain value can be specified using the *has responsible* attribute.

Figure 20 shows a simple *Value structure* where a "Shirt" is decomposed into several other values. The "Shirt" is a marketed value, which is indicated by the blue outline. In this model three choices have to be made: 1) what material to use (one of "Tencel" and "Cotton"), 2) if "Embroidery" should be used (indicated by the dashed line) and 3) which colour to use (one of "Black", "Green" and "Red"). All of those choices should be decided by the customer, which is indicated by the orange

colour of the *has* relation. In any case the Values of "Sewing" and "Local product" are always part of the here described shirts. Also the Values "Tencel" and "Local product" are considered competitive from the modeller's point of view. The "Embroidery" value also implies the colour "Black" (shown by the blue text "Black" underneath "Embroidery", which is only visible when the model attribute "Show details" is selected), which means that if "Embroidery" is chosen, then "Black" also has to be used.

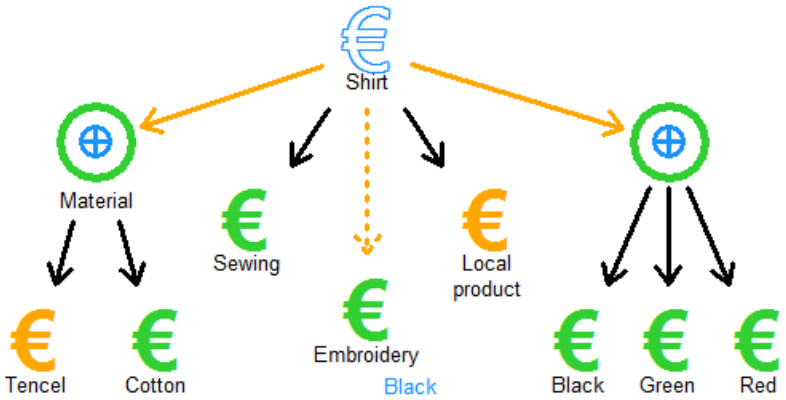

Figure 20: Value structure example

6.1.5 Process model

A *Process model* describes a sequence of activities performed by someone impersonating a role to achieve a certain goal or reach a specific outcome (e.g. solve some issue, create a product etc.). The process should describe what has to happen in order to achieve the goal/outcome, not what could also happen besides it. For this it uses some more or less simple concepts:

- *Process start* – to indicate where to begin.
- *Activity* – to represent a task that is performed.
- *Process end* – to indicate that the process terminates.
- *Sequence relation* – to describe the execution order, they connect two of the other concepts.

Any process model should not have more than one *Process start* and one *Process end* and should only contain *Activities* which are related to the process. Also none of the above concepts should have multiple outgoing *Sequence relations*. Two additional concepts are provided to control the sequence flow in a process:

- *Decision* – representing the need to decide on something and depending on the outcome to choose one specific outgoing path.
- *Hub* – represents a point where several paths can be taken in parallel (splitting hub) and where the paths have to be synchronized again (joining hub). The differentiation between those two is made through the amount of incoming and outgoing sequence relations.

The safest way to think of those is that only one path is taken after a *Decision* similar to an exclusive OR and all paths are taken after a *Hub* similar to an AND. The execution order of paths after a *Hub* doesn't matter and if the necessary resources are available they can also be performed at once. The sequence of those individual paths executed in parallel still has to be adhered to. Unlike the other concepts both the *Decision* and the *Hub* can have multiple outgoing *Sequence relations*[10]. Also the "*Transition condition*" of a *Decision's* outgoing *Sequence relations* should be specified in order to know which path to take depending on the choice made. A correct process using *Decisions* and *Hubs* properly can be seen in Figure 21. In the example also *Swimlanes* are used, which allow to structure

[10] Still a single *Hub* should not have both multiple incoming and multiple outgoing *Sequence relations*. In such a case just use two *Hubs* after another.

the model in a certain way (e.g. by performing roles). The *Swimlane* however only has a meaning to the reader of the model.

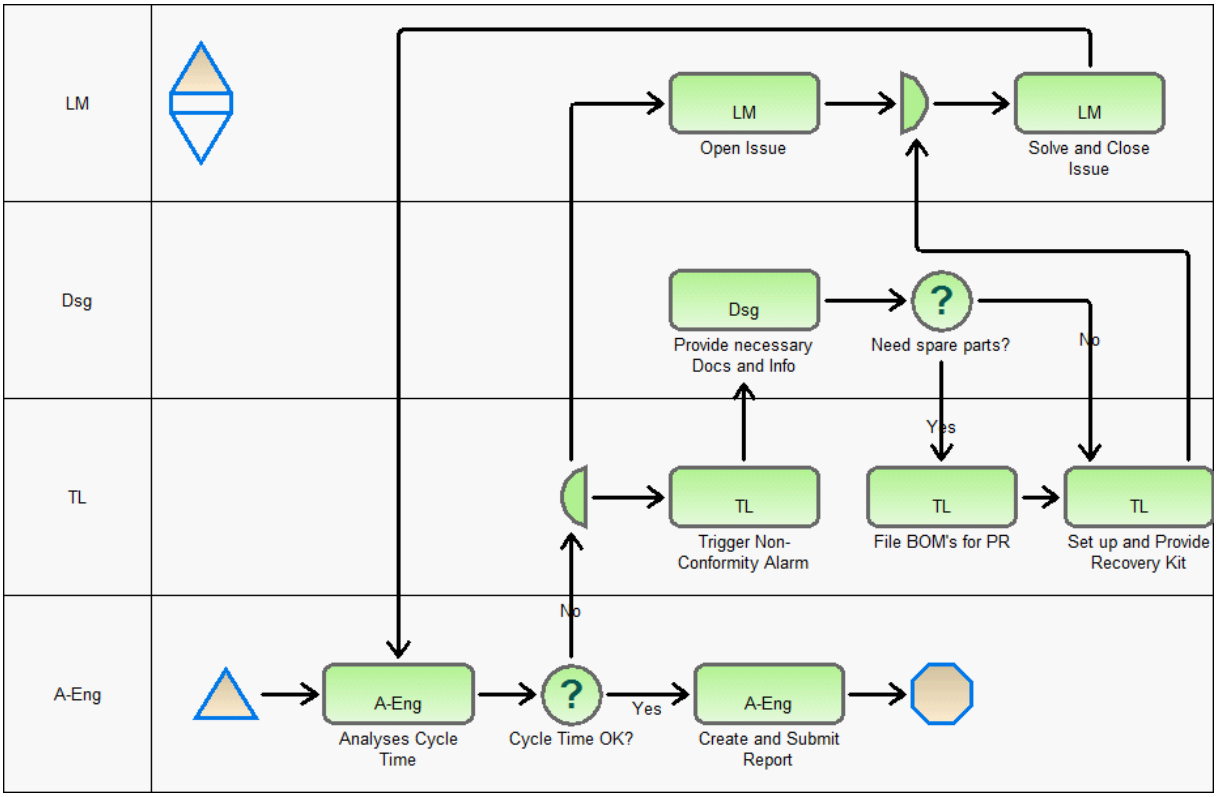

Figure 21: Example WP6 process of multiple roles collaborating

In order to create *Process models* that can further be processed some rules have to be considered. First, for each splitting Hub (i.e. one with multiple outgoing sequence relations) there has to be one corresponding joining Hub (i.e. one with multiple incoming sequence relations). In other words *Hubs* must be used in pairs and each *Hub* can only be part of one such pair. The elements and paths between the splitting and joining *Hub* (the *Hub* pair) are considered to be inside the parallelism. Second, there should also be no *Sequence relations* that jump between the different paths, especially not between one path inside and one path outside of a specific parallelism. Figure 22 shows one correct example and two wrong examples which break one of the above rules.

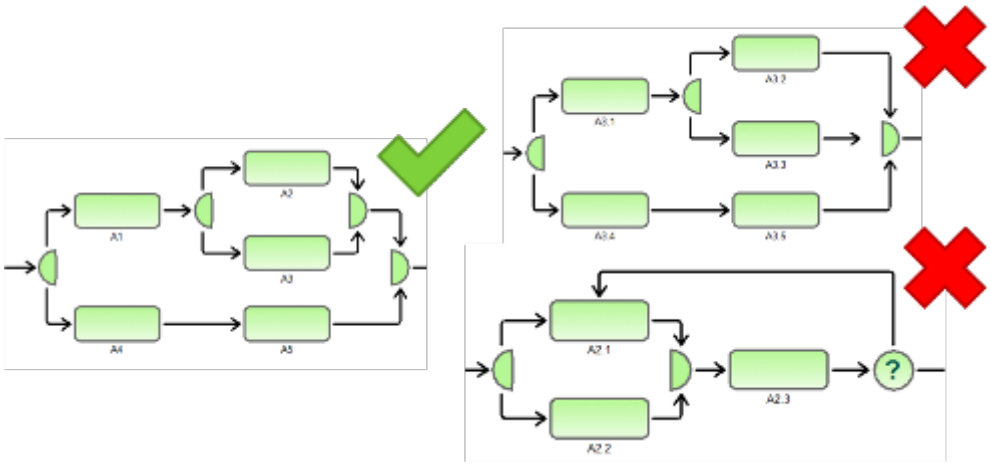

Figure 22: Right and wrong parallelisms

The cases for an exclusive decision and for a parallelism are covered by the *Decision* and *Hub* elements. However, there is still the case that not all but only some paths should be taken in a parallelism, similar to an OR. This can be achieved by combining *Hubs* and *Decisions*. Simply put a

binary decision at the beginning of the parallelism path (i.e. right after the splitting *Hub)* and make one of its outgoing *Sequence relations* go to the corresponding joining *Hub*. An example for this can be seen in Figure 23. The decision does not necessarily need to lead directly to the joining *Hub*, but can also jump only over some activities as long as the above rules for the parallelism are adhered to. The benefit of this is that it rigorously describes what should happen and avoids the case where none of the paths of an OR would be taken.

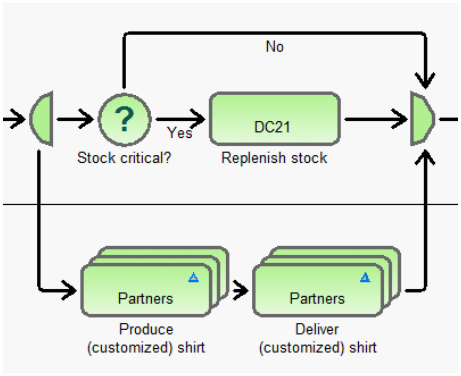

Figure 23: Example of modelling a rigorous OR

Still, even with all those different ways of controlling the sequence flow, the major parts of the process are the activities. At the very least the name of an activity should indicate what is happening, therefore it should at least contain a verb and additional words providing context to the verb (e.g. “process order”, “execute heat test”, “analyse cycle time”). A *Role* is assigned to an activity through the “*Assigned role*” attribute and indicates who is responsible for the activity. If the activity is not decomposed further then it is also assumed to be performed by the assigned role. If possible the roles should be assigned to the activities. Activities can also be described using many different attributes and references, like the time necessary to perform the activity, the cost associated with their execution, the different resources needed for the execution and the capabilities an app should provide among others. If necessary the activities of a process can also be further decomposed and described by other processes by using the *Referenced process* reference.

The collaboration between different organizations can be described through the roles assigned to activities. Figure 21 shows a process where four different roles are collaborating to achieve a common goal, in this case the correct cycle time for a machine. It has both parts where activities are performed in parallel by different roles (e.g. “Open Issue” by the role “LM” and the activities of the role “TL”) as well as activities that cannot be performed until someone else has finished their work (e.g. “File BOM’s for PR” by the role “TL” cannot be performed until “Dsg” finishes “Provide necessary Docs and Info” and decides whether spare parts are needed).

In ComVantage the *Process model* can be used to describe processes as four different views:

- Business view – focuses on activities and their sequence which are necessary to achieve a goal (i.e. the result). The *Business view* should concentrate on what the human has to do and avoid technical details about the execution (e.g. visualize data). An example can be seen in Figure 21.
- Production line – describes what activities are executed on a production line. In general this focuses on the steps necessary for creating a certain *Value* (e.g. a product).
- App requirements view – focuses more on the App requirements and sequence. It can therefore look more like a series of service (or app) calls. It can often be created by splitting and merging activities from the corresponding *Business view*. The *Mobile IT support* reference should be used to indicate which activities are supported by which apps. An example can be seen in Figure 24, where most activities reference a *Mobile IT support feature* (indicated by the small black rectangle in the top left of the activity).

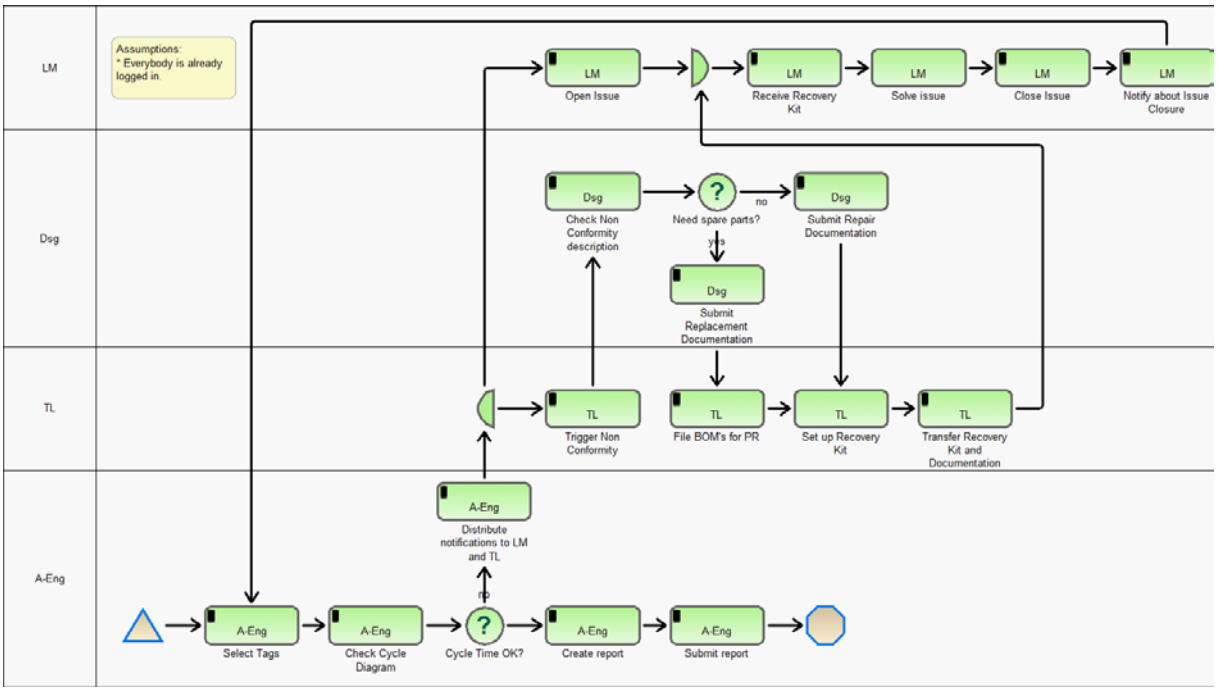

Figure 24: Example WP6 process as an App requirements view

- Interaction view – describes how an app or app orchestration should be interacted with. The result of the process should be the same result as intended by the app it is for. The activities can either be interactions which are happening between the user and the app or functions which are executed without the user (set through the *IF Type* attribute). The *Source type* attribute can further specify for functions where they are executed. Each activity should also use the *Used POIs* table to reference through which POI where data is being received. An example can be seen in Figure 25, where the activities link to *POIs* of a *Mobile IT support model* (see section 1.1.2.1.6). While the *Mobile IT support model* is focusing on the structural view of the app, processes in the *Interaction view* cover the behavioural part of the app.

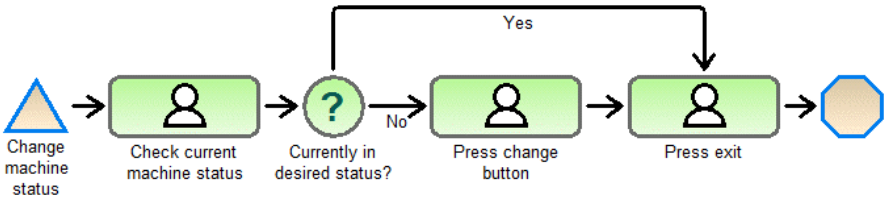

Figure 25: Example for an Interaction view process

The view of a *Process model* can be set in its model attributes. Preferably a single process should not mix views, since otherwise it can get confusing to the person reading the model. Also when referencing a process from an activity, the processes view level should be on the same level as the activities or a "lower" one. For this the order from highest to lowest views is 1) Business view, 2) Production line, 3) App requirements view and 4) Interaction view. If an *App requirements view* has to be described on the same granularity as the *Business view*, then the *View link* element can be used to create a link from the process in the *Business view* to the process in the *App requirements view* (or Interaction view). For example the *View link* element (the rhomb/diamond shape) at the top left in Figure 21 links to the process depicted in Figure 24.

6.1.6 Mobile IT support model

The *Mobile IT Support* model describes interaction requirements. It can be defined in two different views, depending on the value of the *Abstraction level* model attribute:

1. *Abstract UI* – Describes a mobile app in terms of requirements for abstract app interaction. This view is independent of any interaction modality (e.g. haptic, voice, etc.) or visual appearance of the single components of the mobile app. It should suggest the app components and how they can trigger each other, thus giving a structural view on the app (to be complemented by the dynamic view of the interaction process[11] indicating interaction steps for each element).
2. *Concrete UI* – Describes a mobile app through mapping the abstract UI on a concrete visualization for a certain user context and interaction modality, in this case haptic modality. The visual appearance (e.g. size, position, colour etc.) of the single components is relevant at this level in order to provide a suggestive emulation of the user interface, mainly for purposes of requirements validation. It is not intended to generate concrete UI code, as the focus of ComVantage is placed on app orchestration.

The visual difference between those two abstraction levels can be seen in Figure 27 and Figure 28. In order to model the design requirements for a certain mobile app the developer is provided with two major concepts:

Point of interaction (POI) represents the most important part of the *Mobile IT Support* model for the developer of a mobile app. Through this concept it is possible to define different interactions with the user by selecting values in the "*Sub-types"* part of the notebook. The interactions are divided into two types, called *Readable* and *Interactive*. A readable *POI* indicates that it is used by the app to interact with the user (app provides information to the user, e.g. through text, an image, playback of a sound etc.) while an interactive POI provides the possibility for the user to interact with the app (user provides information to the app, e.g. through entering text, taking a picture with the camera etc.). It is also possible for a POI to be both readable and interactive (e.g. a text field that contains the users current bio, but can also be used to directly edit it). Since the exchange of information in either direction is happening through data, it is also possible to specify what data type is used (e.g. "Data" in general, an "Event" like a button press, "Text", "Number", "URI" etc.). The POI also has some attributes in the "*Presentation*" part of the notebook, which allow influencing the representation when a model is viewed in the *Concrete UI* mode.

The concept of *Component* is used to represent a set of *POIs* that are grouped together, which is realised through decomposition. This decomposition is achieved by putting the *POIs* inside the *Component* in the tool. It is also possible for *Components* to contain other *Components*. A typical application for *Components* is to contain everything that should be available on a screen. This is however not set in stone and can be changed through the *Fragmentation level* attribute, where "Generic" can be used to indicate a component not set for any specific device and "Emergent component" to indicate components that emerge in special cases (e.g. like popups). It is not always necessary to further decompose the *Component*. For example it could be specified that a certain component represents "Google Maps" (e.g. through the *sameAs link* attribute) and since the look and feel of "Google Maps" is fixed by an outside actor there is usually little need to further describe it. *Components* can also be repeatable, meaning that all of its contents can repeat multiple times and all of them the same number of times. An example would be a table (the Component set to repeatable) containing sensors (a readable POI) and their values (also a readable POI) for a machine, where the amount of rows is dynamic depends on the type of machine shown. The *is* attribute is also available to foster the reuse of *Components*. For example one *Component* could describe a menu or represent corporate identity elements that are used on several screens (i.e. in several other *Components*).

Figure 26 shows a simple example for a *Component* "Machine status screen" that is meant to display a list of machines and their status (on or off) and contains another *Component* "Machines table" and an interactive *POI* "Exit". The "Machines table" contains a readable *POI* "Machine name and status"

[11] A *Process model* in *Interaction view*.

and an interactive *POI* “Switch status”. It is set to repeatable (as indicated by the R in its top left corner), since the amount of machines that are shown can vary and is unknown during modelling.

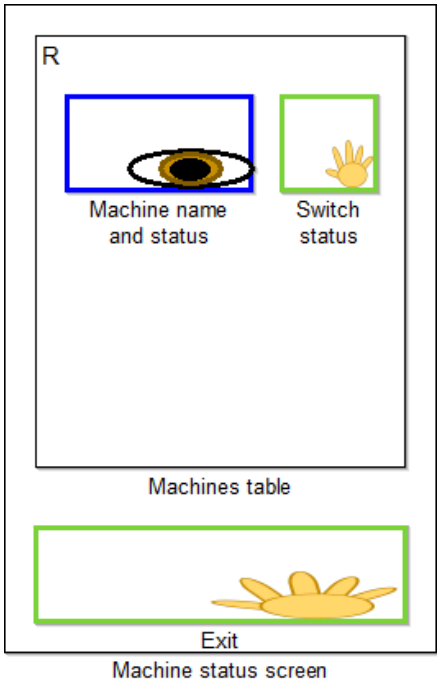

Figure 26: Example for a Component containing another Component and POIs

In order to represent the navigation (from one screen to another or by showing a new *Component*) the *triggers* relation is used. While in most cases the *trigger* relation originates from an interactive *POI*, it can also start from a *Component*. It is possible to specify conditions for the triggers, if for example a button can lead to different screens depending on the users input or if they are logged in or not. Additionally the navigation patterns (see D5.2.3) used or available for a transition can be added to the triggers relation.

By containing the capability description of mobile apps (*Mobile IT support features*) as well as the triggers between those, the *Mobile IT Support Model* implements both the *Navigation map* and a part of the *Mobile support structure* from D3.1.2 (the other part is implemented in the *Resource pool*). As previously stated there are two different view on a *Mobile IT support model*, which can be switched through the model attribute *Abstraction level*. Both Figure 27 and Figure 28 show the same *Mobile IT support model*, however one from the “Abstract UI” view and the other from the “Concrete UI” view. While the notation of POIs in the “Abstract UI” view is only influenced by whether they are interactive and/or readable (e.g. Actuator value is both interactive and readable), the notations in the “Concrete UI” view also consider the specified data type. The black dashed arrows represent the *triggers* relation, with the text above them indicating the navigation patterns.

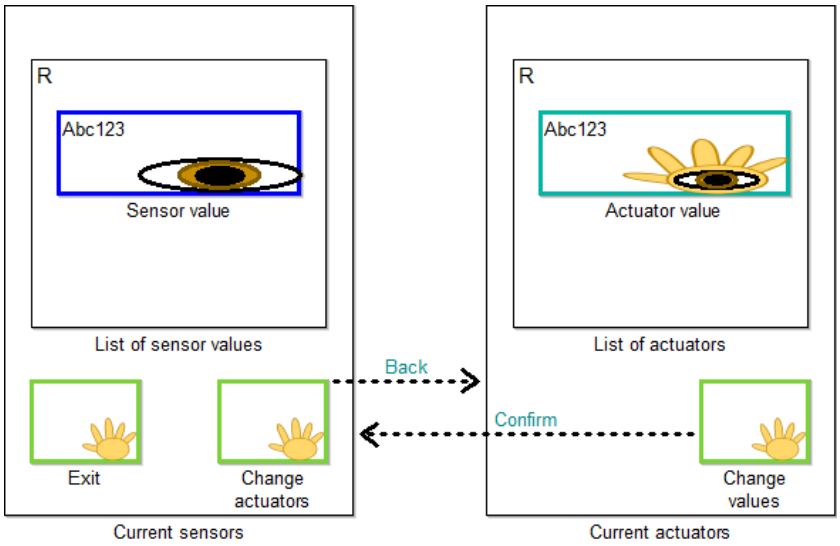

Figure 27: Example of a Mobile IT Support model in the Abstract UI view

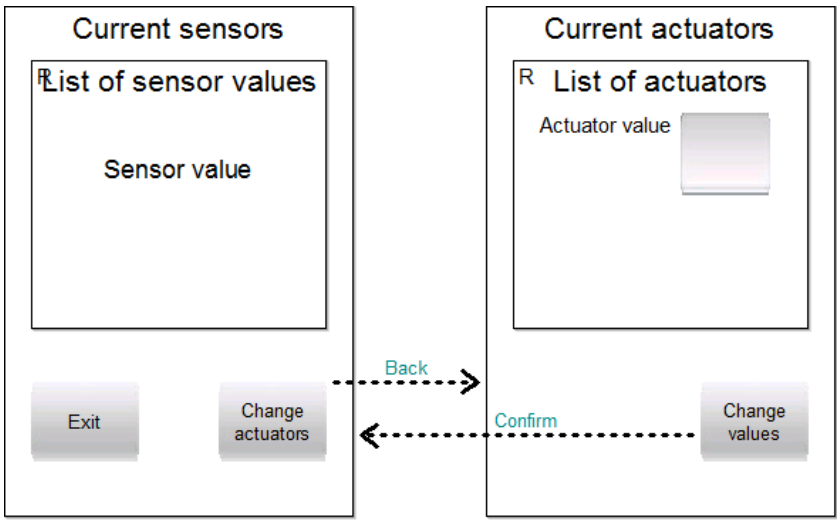

Figure 28: Example of a Mobile IT Support model in the Concrete UI view

6.1.7 Orchestration model

The *Orchestration model* is used to describe the sequences of resource usage for a certain process. In this prototype it currently supports the orchestration of *Mobile IT support features* which can then be used by an app orchestration.

The two general concepts in the *Orchestration* model are the resource and the *Followed by* relation. A *Resource* represents the thing that is used and the sequence is described through the *Followed by* relations, which can contain "*Transition conditions*". In this regard they are similar to the *Activities* and *Sequence relations* from the *Process* model. However it differs from the *Process* model in its intention and how it achieves it. First the orchestration is for a process and is not about executing activities, but using and reusing resources. Therefore it is not focusing on activity execution time, activity execution costs, the proper interaction with resources or what multitudes of different resources are used in a single step, but instead focuses on the resources used and their sequence.

Second the orchestration is concentrating on the resource usage for a certain collaboration partner, meaning it is from this collaboration partner's view. However to properly depict the collaboration it also uses the additional concepts of *Suspension points* and *Notifications*. *Suspension points* allow specifying a point in the sequence where some message or notification is necessary to continue. They are defined by two types: "Entry" and "Dependency". An entry type *Suspension point* indicates that

the orchestration can be entered here when a certain notification is received, while a dependency type *Suspension point* specifies that a notification containing additional data is necessary to continue further. *Notifications*, which are handled through table attributes in the *Orchestration*, allow describing what should be sent and, if applicable, also towards which activity. Those notifications as well as the ones used in the *Suspension points* are modelled through *Information resources* in the *Information space model (see section 1.1.2.1.9)*.

Additionally the *Synchronization point* and the *Path split* are available to further detail the flow in an *Orchestration model*. Their *Control flow pattern* attribute can be used to specify if they represent a "Synchronization/Parallel split" or a "Simple merge/Exclusive choice". The *Resources* also contain a *Control flow pattern* attribute in order to preserve older models. The details on how splitting and synchronizing (merging) paths works is dependent on the used orchestration engine.

An example for an *Orchestration model* where the sequence of *Mobile IT support features* for the A-Eng role can be seen in Figure 29 (the corresponding process can be seen in Figure 24). Both the left side and the right side of the figure represent the same orchestration, however the left side uses an explicit "Exclusive choice" *Path split*, while the right side makes use of the backwards compatibility. In this case instead of looping back from "Notification distribution" to the beginning it is assumed that the orchestration will simply be executed a second time when a "check cycle time" notification is received.

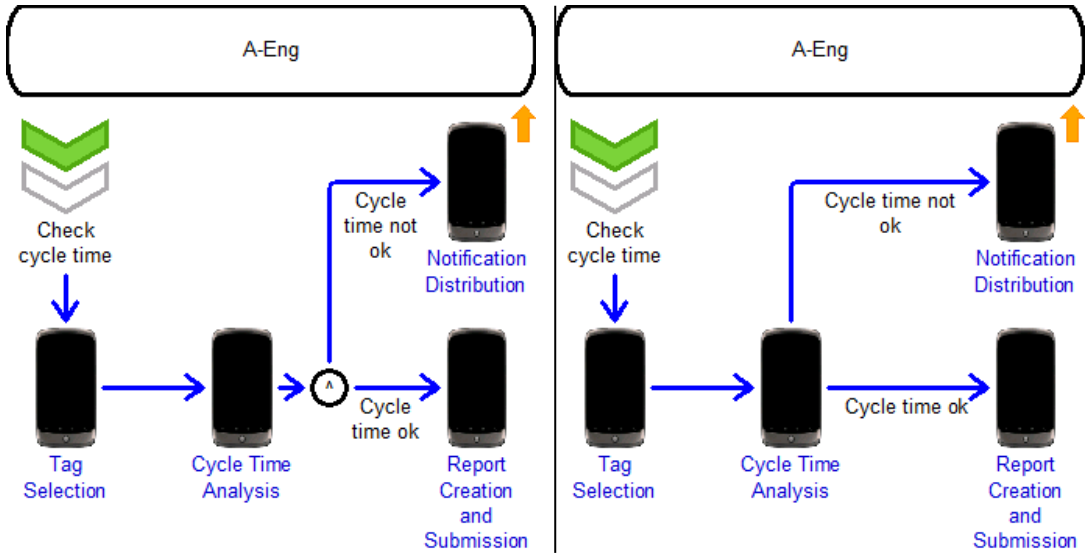

Figure 29: Example of the same orchestration for the A-Eng role from process in Figure 24 in two different models

The notations of the *Resources* in an *Orchestration model* can be adapted through the *Show* model attribute to either show the apps referenced by the elements (dark blue text), the names of the elements (black text in parentheses; could be the names of the activities the resource usage is based on) or both.

6.1.8 Location structure

The location structure is used to describe important locations, both physical and digital, and provide information about their decomposition. For this the *Physical location*, *Digital location*, *Plant* and *Production line* classes are available. For each of those the area they depict and whether it is absolute or relative can be specified. For *Physical locations* (this includes the *Plant* and *Production line*) it is recommended that the *Area* attribute uses a format for addresses that is understood by the chosen map application[12] where applicable, while *Digital locations* should specify a URL. When the type is set to "Absolute", then the *Area* attribute should contain all the information necessary to find it,

[12] E.g. Google maps, Bing maps etc.

otherwise if it is “Relative” then the *Area* attribute has to be considered in relation to the location it is part of.

Figure 30 contains a Location structure that has several physical locations, one of which is a *Plant* in “Pomigilano” (Italy) and the other a *Production line* of that plant. It also contains three digital locations, one representing the ComVantage homepage, which contains a sub-page with the public deliverables and the third a digital location where orders are stored.

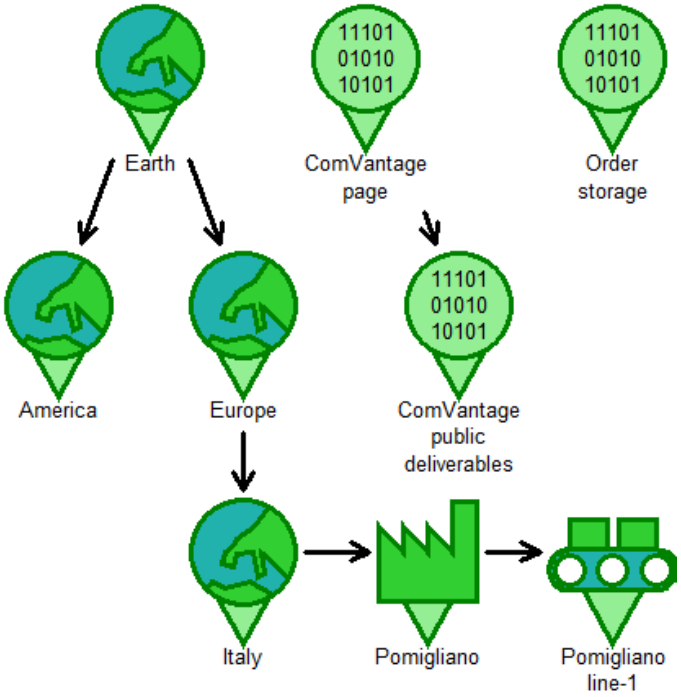

Figure 30: Example Location structure with physical and digital locations

6.1.9 Information space model

The *Information space model* is used to describe information on different levels of abstraction, how it can be accessed and, if applicable, how they are decomposed. Generally information is depicted through *Information resource* objects, which can be described in further detail through the *Entity*, *Relation* and *Attribute* objects and linked using the *has capability* relation.

Information resources represent a collection of meaningful information and can be categorized in “First hand information” (i.e. mostly raw and unprocessed), “Refined information” (i.e. processed and enriched) and “Aggregated information” (i.e. assembled from different sources), which assumes that it is also refined. Additionally for each *Information resource* it can be specified if it should be public, restricted or private. This is however not a replacement for the proper description of access control requirements, which can be specified using the *Access control requirements* table attribute. More details about this can be found in section 1.1.2.2.2.

The *Entity*, *Relation* and *Attribute* should be understood similarly to the elements of an Entity-Relationship diagram. The *relates* relation should be used to link a *Relation* with the *Entities* and can specify the cardinality and a role for the targeted *Entity* to give it some context. *Attributes* can be linked to the *Entities* and *Relations* through the decomposition relation, which can state if the decomposition is separable or not. If the decomposition is set to be separable (dashed line) then the entity/relation can be available without those attribute. Otherwise (if it is not separable), whenever the entity/relation is available the attribute also is available[13]. The *relates* relation is always inseparable, meaning that the relation cannot be without the entities.

[13] Note that an inseparable *Attribute* does not necessarily mean it is part of the “Primary key” in case of a relational model.

Additionally *Information access* elements can be used to describe what access to the information is available. It describes the finer details of the information storage. As such the information can be available on different types media, here categorized as “Paper-based”, “Paperless” or as “Linked data”, which is described through the *Media type* attribute. This differentiation is of interest for security reasons since tangible information has to be accessed and secured differently than intangible information. Also each *Information access* can provide the information from different sources (e.g. local, remote server etc.) and allow different types of operations on the information, which are generally classified as “Create”, “Read”, “Update” and “Delete” operations. In case the information is accessed from a remote location, the *executed on* table attribute can be used to specify from which digital location (see section 1.1.2.1.8) to access the information and, if necessary, what query to use.

An example for an Information space model can be seen in Figure 31. It contains three *Information resources* (“Customers”, “Contacts” and “Orders”). Two *Information accesses* are available for “Order”, one depicts read access to *Linked Data* and the other read access to paperless data. Additionally some of the data structure is described through the Entities (“Order”, “Person” and “Product”), Relations (“has ordered” and “contains”) and Attributes (“Name”, “Address”, “Order ID” etc.). What is not shown in the figure is that there is a connection between the *Information resources* and the *Entities*/*Relations*/*Attributes*, which is handled through the *has capability* attribute. For example the “Orders” *Information resource* has the capabilities “contains”, “has ordered” and “Order”, describing what information is provided. Through the way the model is described, this also means that the “Orders” also includes the “Person” and “Product” entities (because they are inseparable from the “has ordered”/”contains” *Relations*) and all inseparable *Attributes* of the *Entities* (e.g. “Order ID”, “Delivery address”, “Email” etc.)

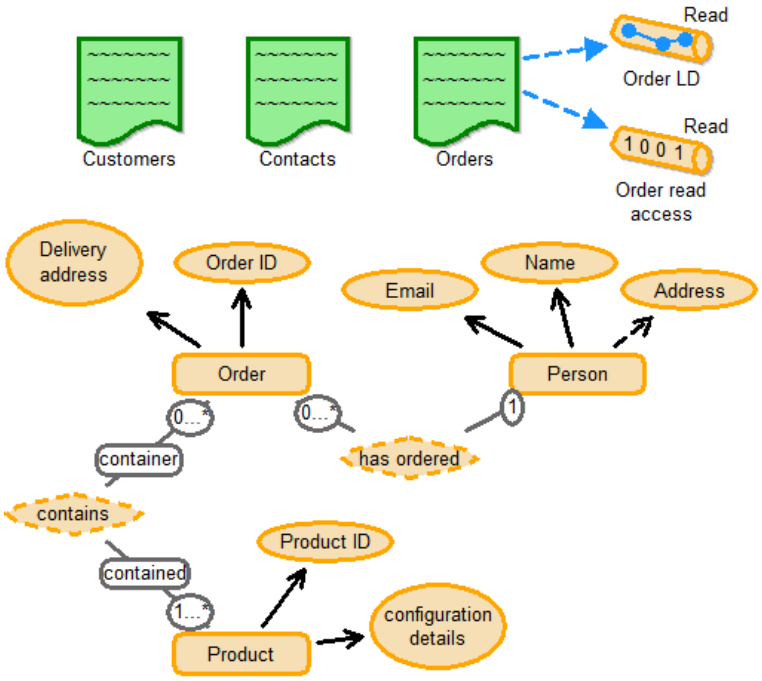

Figure 31: Information space model example

6.1.10 Station structure

The *Station structure* is used to describe stations that are typically used in a production line and how they are decomposed into different parts and sensors. The structure of a *Station structure* model is very similar to that of a *Machine state model* (see section 1.1.2.1.11). In this model the stations are described on a type level, which is covered through the *Station*, of which there can be several instances, which are depicted through *Station instances*.

The *Station* can be further decomposed into *Station parts*, which can be one of several sub-types ("Robot", "Safety device" etc.) and can be decomposed into other *Station parts*. It is also possible to attach *Sensors* to *Stations* and *Station parts*. *Sensors* allow retrieving data about the thing they are attached to. It is possible to specify for them the physical property it is measuring, the unit it is measured in and the transient interval. The *retrieve from* table attribute of a *Station instance* allows specifying endpoints from where sensor values for the specific instance can be retrieved. *Stations*, *Station parts* and *Sensors* can also reference a *Component type* from a *Resource pool* through the *of type* attribute.

Additionally *Machine defects* are available to describe states that are out of the ordinary. They can further be detailed by describing general *Sensor states* (e.g. "Safe", "Critical", "On") in the *Defect description* table attribute, to facilitate the automatic discovery of defects. It is also possible to describe the required set of skills/knowledge and the required proficiency in those to solve a defect in the *Skills/Knowledge* table attribute. The *for types* attribute allows to specify for which types of *Stations* and *Machine types* the defect can occur, while the *Recommended approach* attribute can link to a process that details how to solve the defect. In the *Station structure* it is also possible to decompose a *Station* into *Operators*, to indicate that a human is working on the station or supervising it. Those Operators should further link to a *Role* from the *Business and Organization structure*.

Figure 32 shows a *Station structure* that contains a "Welding station", of which two instances ("WS-1" and "WS-2") are available. The welding station contains two *Station parts*, one of which is a "Transportation belt" that has the *Sensor* "Speed" available. One defect "Broken belt" is described, which occurs when the "Speed" sensor has the status "at risk" (not seen in the figure).

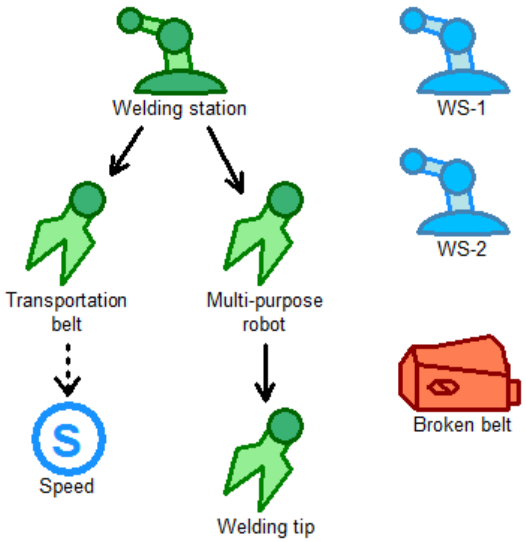

Figure 32: Station structure example

6.1.11 Machine state model

The *Machine state model* is used to describe machines and how they are decomposed into their parts, sensors and actuators. The structure of a machine state model is very similar to that of a *Station structure* model. In this model the machines are described on a type level, which is covered through the *Station*, of which there can be several instances, which are depicted through *Station instances*.

The *Machine type* can be further decomposed into *Machine parts*, which in turn can be decomposed into other *Machine parts*. Both *Machine types* and *Machine parts* can be described through their estimated lifetime, their general reliability and can link to processes describing how the *Machine type*/Machine part should be tested. It is also possible to attach *Sensors* and *Actuators* to *Machine types* and *Machine parts*. *Sensors* allow retrieving data about the thing they are attached to, while *Actuators* represent a part that allows controlling the thing they are attached to (e.g. control the

rotation speed). It is possible to specify for them the physical property they are using, the unit it is measured in and the transient interval. The *retrieve from* table attribute of a *Machine instance* allows specifying endpoints from where sensor values for the specific instance can be retrieved. *Machine types*, *Machine parts*, *Sensors*, and *Actuators* can also reference a *Component type* from a *Resource pool* through the *of type* attribute.

Additionally *Machine defects* are available to describe states that are out of the ordinary. They can further be detailed by describing general *Sensor states* (e.g. “Safe”, “Critical”, “On”) in the *Defect description* table attribute, to facilitate the automatic discovery of defects. It is also possible to describe the required set of skills/knowledge and the required proficiency in those to solve a defect in the *Skills/Knowledge* table attribute. The *for types* attribute allows to specify for which types of *Stations* and *Machine types* the defect can occur, while the *Recommended approach* attribute can link to a process that details how to solve the defect.

Figure 33 shows a *Machine state model* that contains a “Printer”, of which two instances (“P-1” and “P-2”) are available. The printer contains two *Machine parts*. The “Toner” part has a *Sensor* “Remaining charge” available, while the “Feed roll” part has a “Roll speed” *Actuator*. One defect “Empty toner” is described, which occurs when the “Remaining charge” sensor has the status “critical” (not seen in the figure).

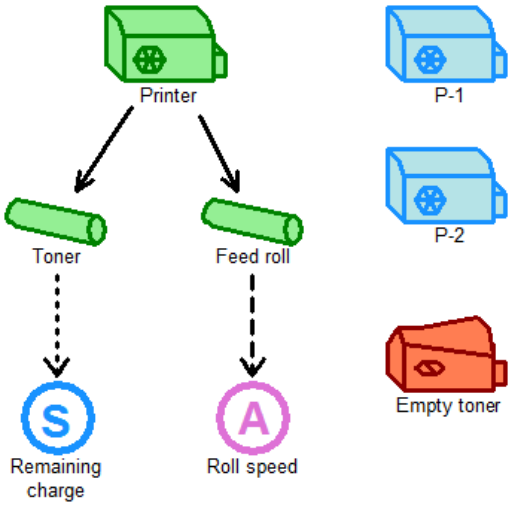

Figure 33: Machine state model example

6.2 Scenario Independent Modelling Application Guidelines

Several results can be achieved by creating models in the current prototypical implementation. What exactly they are and how to achieve them is described in the sections 1.1.2.2.1 to 1.1.2.2.5. Those procedures represent guidelines and as such it might be useful to stray from the here provided path depending on the case. This however has to be decided on a case to case basis. Modelling does not always follow a rigid procedure. Sometimes creating a certain model provides new insights on the topic that requires changing previously created models. For example when modelling a process one might realize that a certain *Role* is missing in the *Business and Organization structure*. Therefore going back and forth between procedure steps is acceptable.

6.2.1 Model Export as RDF

It is recommended that model information is exposed via the developed RDF export to the Linked Data environment in order to enable the development of what could be called “model aware information systems” – that is, applications that make use (i.e. query) model information at run-time. A proof-of-concept for this type of system has been developed in ComVantage in the form of the Industrial App Framework, where app orchestrations are guided by the information described at design time by stakeholders in their business process models, based on their business view. Through

this, the models or their parts can further be linked to other descriptions and elaborate queries can be executed by using for example SPARQL. The general procedure to accomplish this is as follows:

1. Create models in the prototype (details in sections 1.1.1.1 and 1.1.2.1)
2. Export the desired models as XML (details in section 1.1.1.3)
3. Use additional RDF Export files which can be obtained from the OMILab portal after registering. Further up to date details on how to use the RDF Export can be found together with example files in the archive available there[14].

An overview of how the current RDF Export user interface looks can be seen in Figure 34.

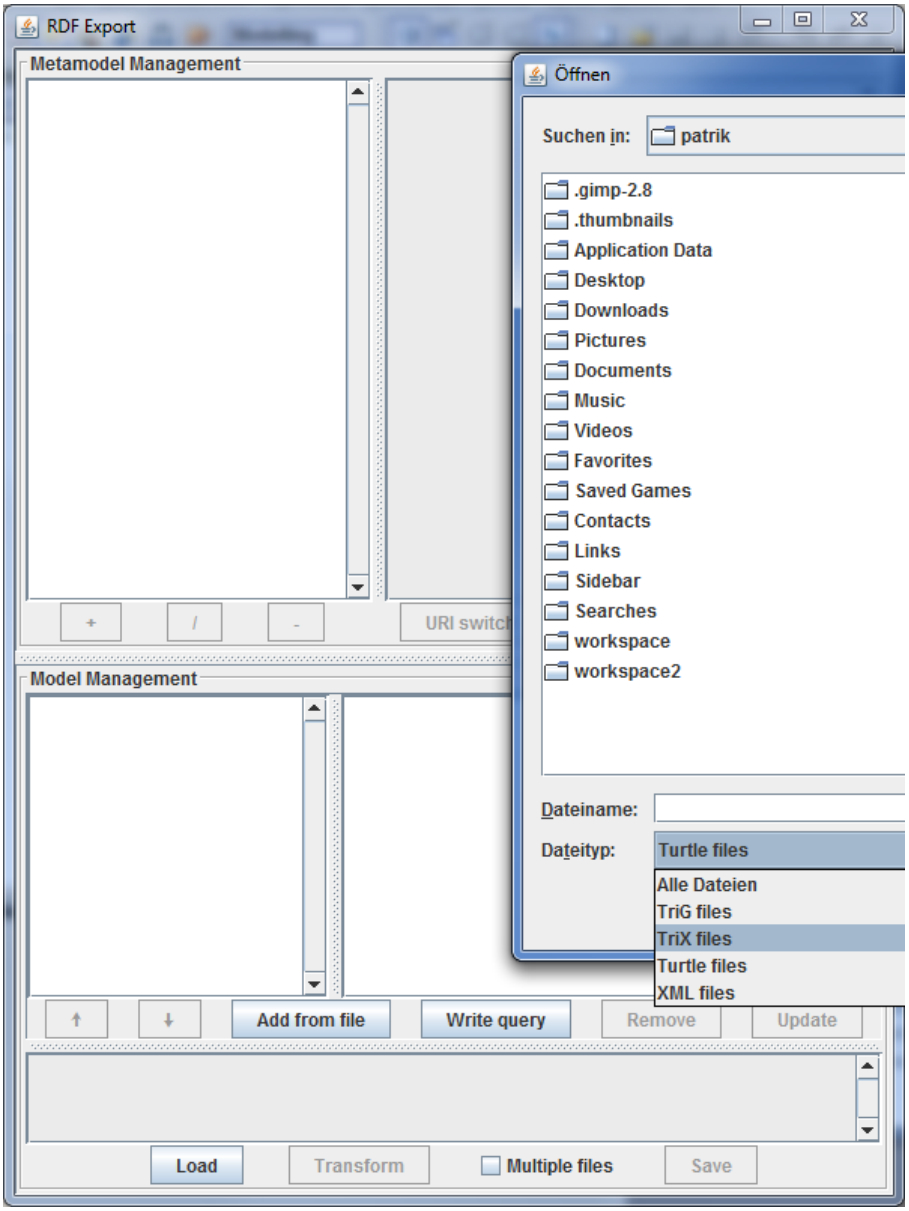

Figure 34: RDF Export Graphic User Interface

Figure 35 shows an example for a *Process model*, a *Business and Organization structure* and an *Information space model* and Code 1 contains the resulting RDF in TriG syntax. Some unnecessary attributes (e.g. for debugging or only for visualization purposes) have been removed from example RDF to reduce the amount of space needed. In Code 2 an example query asking for all *Roles* that

[14] http://www.omilab.org/web/comvantage/home

need access to an *Information resource* can be seen[15]. The query example also covers the role hierarchy. Further details about the RDF Export structure and examples can be found in D3.1.2 and the metamodel description provided with the RDF Export files.

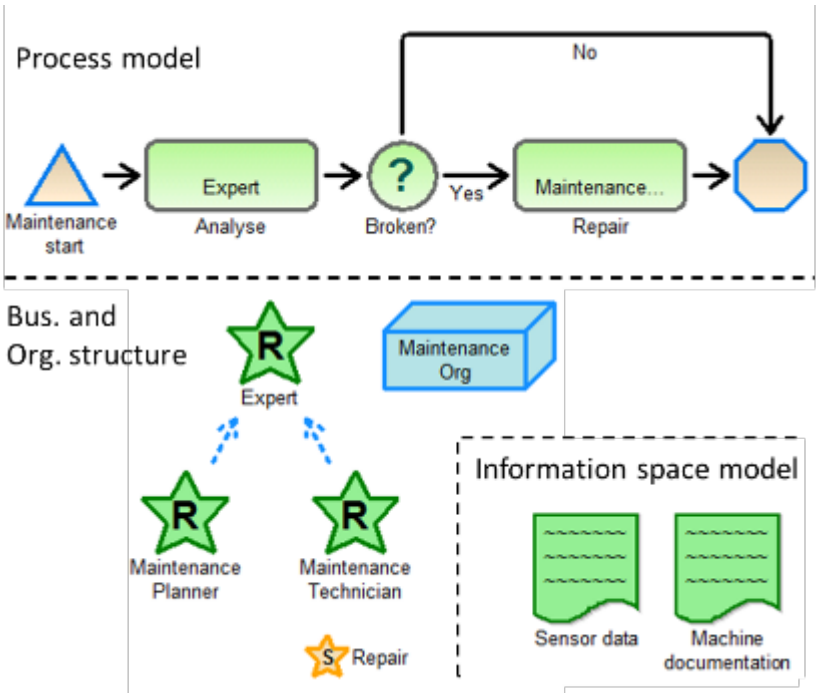

Figure 35: Example process and resource pool

```
@prefix :        <http://www.comvantage.eu/example#> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix cv:      <http://www.comvantage.eu/mm#> .

:Business_and_Ogranization_structure-Maintenance_Org_ {
    :Role-507710-Expert
        a       cv:Instance_class , cv:Role ;
        cv:Name "Expert" .

    :Activity-507730-Repair
        cv:Assigned_role :Role-507707-Maintenance_Technician ;
        cv:described_in :Process_model-Maintenance_Process_ .

    :Business_entity-507701-Maintenance_Org
        a       cv:Instance_class , cv:Business_entity ;
        cv:Name "Maintenance Org" ;
        cv:described_in :Business_and_Ogranization_structure-Maintenance_Org_ ;
        cv:owns :Role-507704-Maintenance_Planner , :Role-507707-
Maintenance_Technician .

    :Role-507704-Maintenance_Planner
        a       cv:Instance_class , cv:Role ;
        cv:Name "Maintenance Planner" ;
        cv:described_in :Business_and_Ogranization_structure-Maintenance_Org_ ;
```

[15] Tested using the openRDF server Sesame and its workbench 2.7.9.

```
        cv:is_a :Role-507710-Expert .

    :Role-507707-Maintenance_Technician-SkillsKnowledge_1
        cv:Element :Skill-507715-Repair ;
        cv:Level "3" ;
        cv:described_in :Business_and_Ogranization_structure-Maintenance_Org_ .

    :Skill-507715-Repair
        a       cv:Skill , cv:Instance_class ;
        cv:Name "Repair" ;
        cv:described_in :Business_and_Ogranization_structure-Maintenance_Org_ .

    :Role-507707-Maintenance_Technician
        a       cv:Instance_class , cv:Role ;
        cv:Name "Maintenance Technician" ;
        cv:SkillsKnowledge (:Role-507707-Maintenance_Technician-SkillsKnowledge_1)
;
        cv:described_in :Business_and_Ogranization_structure-Maintenance_Org_ ;
        cv:is_a :Role-507710-Expert .

    :Activity-507727-Analyse
        cv:Assigned_role :Role-507710-Expert ;
        cv:described_in :Process_model-Maintenance_Process_ .
}

:Information_space_model-Maintenance_Information_ {
    :Information_resource-508004-Sensor_data
        a       cv:Information_resource , cv:Instance_class ;
        cv:Access_control_requirements
            (:Information_resource-508004-Sensor_data-
Access_control_requirements_1) ;
        cv:Name "Sensor data" .

    :Information_resource-508004-Sensor_data-Access_control_requirements_1
        cv:Action :Action_type-Use_or_Read ;
        cv:Constraint_description "Maintenance ticket is created" ;
        cv:Subject :Role-507710-Expert .

    :Role-507710-Expert
        cv:described_in :Business_and_Ogranization_structure-Maintenance_Org_ .

    :Information_resource-508007-Machine_documentation
        a       cv:Information_resource , cv:Instance_class ;
        cv:Name "Machine documentation" .

    :Activity-507727-Analyse-Uses_resource_1
        cv:Resource :Information_resource-508004-Sensor_data ;
        cv:described_in :Process_model-Maintenance_Process_ .

    :Activity-507730-Repair-Uses_resource_1
        cv:Resource :Information_resource-508007-Machine_documentation ;
        cv:described_in :Process_model-Maintenance_Process_ .
}
```

```
:Process_model-Maintenance_Process_ {
    :Role-507710-Expert
        cv:described_in :Business_and_Ogranization_structure-Maintenance_Org_ .

    :Role-507707-Maintenance_Technician
        cv:described_in :Business_and_Ogranization_structure-Maintenance_Org_ .

    :Activity-507727-Analyse-Uses_resource_1
        cv:Action :Action_type-Use_or_Read ;
        cv:Resource :Information_resource-508004-Sensor_data .

    :Information_resource-508004-Sensor_data
        cv:described_in :Information_space_model-Maintenance_Information_ .

    :Process_start-507724-Maintenance_start
        a       cv:Instance_class , cv:Process_start ;
        cv:Intention_type "Intended" ;
        cv:Name "Maintenance start" .

    :Decision-507736-Broken
        a       cv:Instance_class , cv:Decision ;
        cv:Question "Broken?" .

    :Sequence_relation-507739-Maintenance_start-Analyse
        a       cv:Relation_class , cv:Sequence_relation ;
        cv:Transition_probability "1" ;
        cv:from_instance :Process_start-507724-Maintenance_start ;
        cv:to_instance :Activity-507727-Analyse .

    :Activity-507730-Repair
        a       cv:Activity , cv:Instance_class ;
        cv:Assigned_role :Role-507707-Maintenance_Technician ;
        cv:Name "Repair" ;
        cv:Uses_resource (:Activity-507730-Repair-Uses_resource_1) .

    :Information_resource-508007-Machine_documentation
        cv:described_in :Information_space_model-Maintenance_Information_ .

    :Sequence_relation-507741-Broken-Repair
        a       cv:Relation_class , cv:Sequence_relation ;
        cv:Transition_condition "Yes" ;
        cv:Transition_probability "1" ;
        cv:from_instance :Decision-507736-Broken ;
        cv:to_instance :Activity-507730-Repair .

    :Sequence_relation-507740-Analyse-Broken
        a       cv:Relation_class , cv:Sequence_relation ;
        cv:Transition_probability "1" ;
        cv:from_instance :Activity-507727-Analyse ;
        cv:to_instance :Decision-507736-Broken .

    :Sequence_relation-507743-Broken-Maintenance_end
        a       cv:Relation_class , cv:Sequence_relation ;
        cv:Transition_condition "No" ;
```

```
        cv:Transition_probability "1" ;
        cv:from_instance :Decision-507736-Broken ;
        cv:to_instance :Process_end-507733-Maintenance_end .

    :Sequence_relation-507742-Repair-Maintenance_end
        a       cv:Relation_class , cv:Sequence_relation ;
        cv:Transition_probability "1" ;
        cv:from_instance :Activity-507730-Repair ;
        cv:to_instance :Process_end-507733-Maintenance_end .

    :Process_end-507733-Maintenance_end
        a       cv:Process_end , cv:Instance_class ;
        cv:Intention_type "Main" ;
        cv:Name "Maintenance end" .

    :Activity-507730-Repair-Uses_resource_1
        cv:Action :Action_type-Use_or_Read ;
        cv:Resource :Information_resource-508007-Machine_documentation .

    :Activity-507727-Analyse
        a       cv:Activity , cv:Instance_class ;
        cv:Assigned_role :Role-507710-Expert ;
        cv:Name "Analyse" ;
        cv:Uses_resource (:Activity-507727-Analyse-Uses_resource_1) .
}

cv:graphmetadata {
    :Process_model-Maintenance_Process_
        a       cv:Model_class , cv:Process_model ;
        cv:Name "Maintenance Process" ;
        cv:Type "Process model" ;
        cv:View "Business view" .

    :Information_space_model-Maintenance_Information_
        a       cv:Information_space_model , cv:Model_class ;
        cv:Name "Maintenance Information" ;
        cv:Type "Information space model" .

    :Business_and_Ogranization_structure-Maintenance_Org_
        a       cv:Business_and_Ogranization_structure , cv:Model_class ;
        cv:Name "Maintenance Org" ;
        cv:Type "Business and Ogranization structure".
}
```

Code 1: Reduced example process and resource pool as RDF

```
PREFIX :<http://www.comvantage.eu/example#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX cv:<http://www.comvantage.eu/mm#>
PREFIX owl:<http://www.w3.org/2002/07/owl#>

SELECT ?name
WHERE {
```

```
    GRAPH ?irrep {
        #1 Get the information resource
        ?ir a cv:Information_resource.
        FILTER (?ir = :Information_resource-508004-Sensor_data)
        ?accr cv:Resource ?ir.
        ?accr cv:described_in ?proc.
    }
    GRAPH ?proc {
        #2 Find the activities which use this and get their role
        ?act cv:Uses_resource ?anonlist.
        ?anonlist rdf:list*/rdf:first ?accr.
        ?act cv:Assigned_role ?assigrole.
        ?assigrole cv:described_in ?rolrep.
    }
    GRAPH ?rolrep {
        #3 Get the role that is assigned to activity from 2
        ?transrole cv:is_a* ?assigrole.
        ?transrole cv:Name ?name.
        # The above two statements actually cover the node itself and the hierarchy
    }
    GRAPH cv:graphmetadata {
        # Ensure that the graphs have the right type (consider it a trial)
        ?proc a cv:Process_model.
        ?irrep a cv:Information_space_model.
        ?rolrep a cv:Business_and_Ogranization_structure.
    }
}
```

Code 2: Example SPARQL query to get all roles for a specific information resource

6.2.2 Process Stepper and Simulation

Another extension available in the current OMI prototype implementation is the process stepper and simulator. It allows stepping through a process model at the users desired pace and providing a reflection of each visited node. Additionally it also provides functionality to simulate processes. The general procedure to accomplish this is as follows:

1. Create models in the prototype (details in sections 1.1.1.1 and 1.1.2.1)
2. Export the desired models as XML (details in section 1.1.1.3)
3. Use additional Process Stepper and Simulator files which can be obtained from the OMI portal after registering[16]. Further up to date details on how to use the Process Stepper and Simulator can be found together with example files in the archive available there.

An overview of how the current Process Stepper and Simulator user interface looks can be seen in Figure 36 and Figure 37 shows the result of a simulation, where the pop-up contains additional information for a selected path.

[16] http://www.openmodels.at/web/comvantage/home

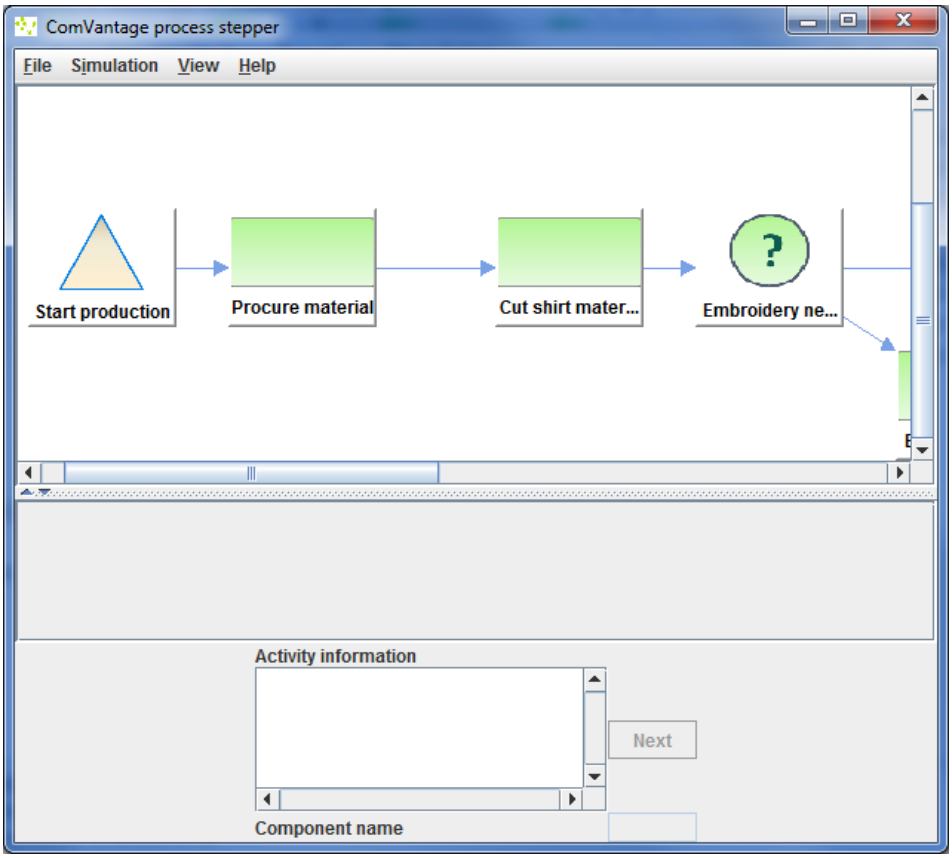

Figure 36: Process Stepper and Simulator GUI

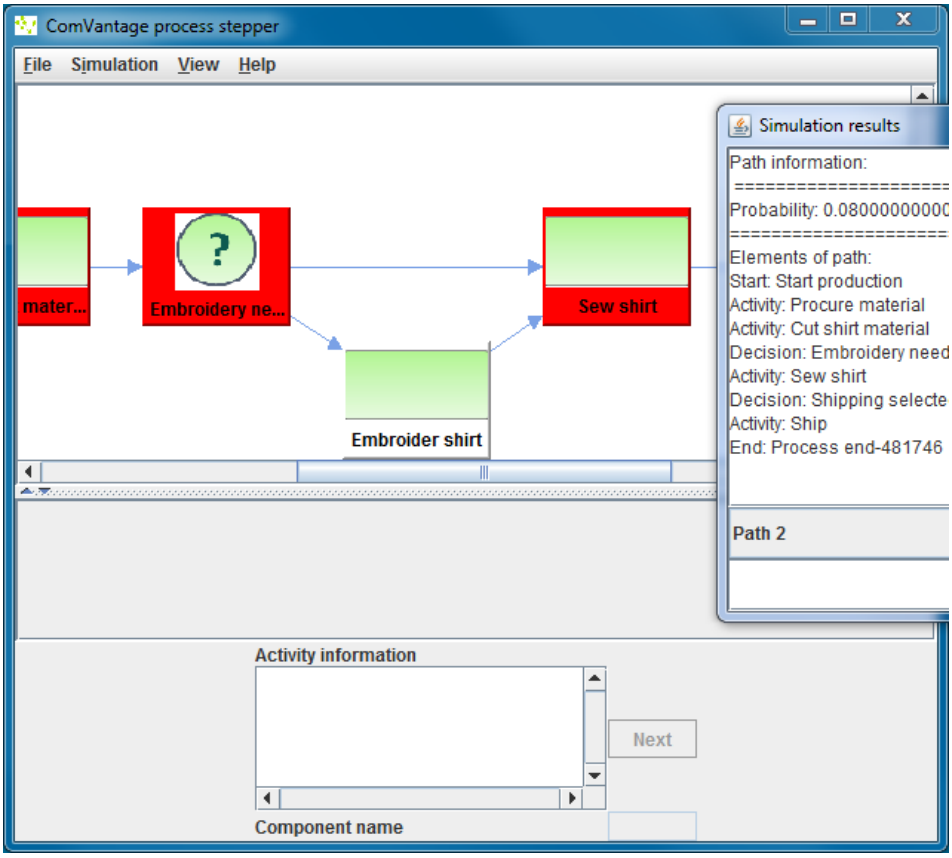

Figure 37: Simulation result example, showing Path number 2 (out of 4)

6.2.3 Modelling Access Control

The current OMI prototype allows modelling two things for access control: access requirements and access policies. The access requirements describe who needs to do what on a certain resource, while the access policies describe who is allowed to do what on a certain resource. Both rely on the concepts of "Subject" performs an "Action" on a "Resource". The recommended approach is to first describe the access requirements and then base the access policy based on those by performing the following steps:

1. Create a *Process model* in *Business view* and describe what has to be done and by whom (i.e. by which *Role*) independent of resources used. The Roles are assigned through the "*Assigned role*" attribute of the *Activity*. Note that unless an *Activity* is further decomposed it will be assumed that the assigned *Role* is both responsible and performing the *Activity*.
2. If necessary, decompose the process and describe it in a finer granularity and/or in a different view. The decomposition of an *Activity* can be done through its "*Referenced process*" attribute. To link a process on the same level but in a different view, use the *View link* element in the *Business view* process and the corresponding reference it provides.
3. Reuse existing resources or create new necessary resources and assign them to the activities where they should be used. The resources are assigned through the "*Uses resource*" table where each row indicates the need to execute an action on a resource. The table contains three columns:
    a. The first column references what action should be performed, covering the "Action" concept of the access control approach. The target should be an *Action type*, which can be created in a *Resource pool*. It is recommended to at least have an *Action type* representing all possible actions, one to represent types of actions that do not alter the resource (e.g. "Use or Read"), one to represent types of actions that alter the resource (e.g. "Change or Write") and to properly describe their hierarchy using the *is a* relation provided. Additional *Action types* can be added to this hierarchy accordingly.
    b. The third column indicates what additional constraints might be in place when the action is executed using natural language.

    The "Subject" portion can be taken from the *Role* assigned to the *Activity* if it is not further decomposed. Additionally the *Activity* itself can be seen as context for the access to the resource. If the *Activity* is decomposed then the process referenced through "*Referenced process*" should contain the details about the necessary access requirements.

Those first steps create the access requirement description. A simple example can be seen in Figure 38 and should be read as: "Expert" (S) needs "Use or Read" (A) access to "Sensor data" (R) [during activity "Analyse"]. An example containing the important RDF statements in TriG syntax can be seen in Code 3.

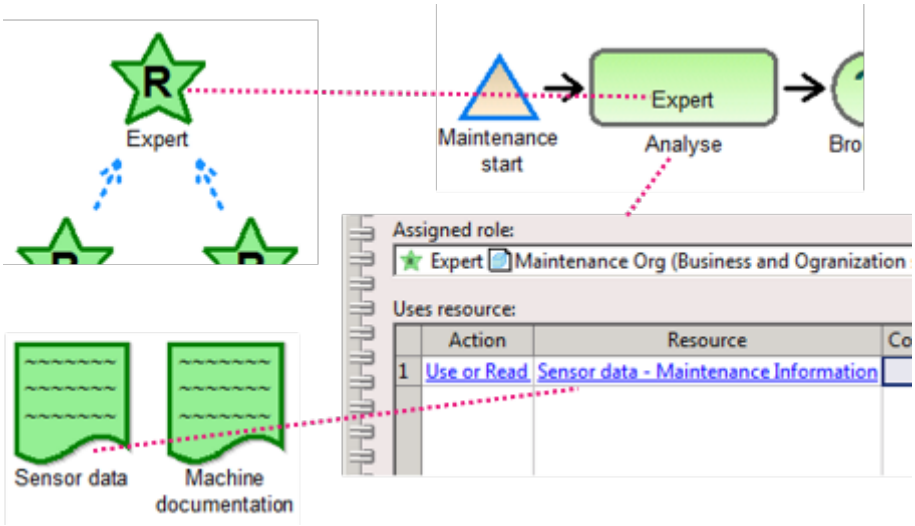

Figure 38: Example showing an access requirement description

```
:Process_model-Maintenance_Process_ {
    :Activity-507727-Analyse
        a       cv:Activity , cv:Instance_class ;
        cv:Assigned_role :Role-507710-Expert ;
        cv:Name "Analyse" ;
        cv:Uses_resource (:Activity-507727-Analyse-Uses_resource_1) .

    :Activity-507727-Analyse-Uses_resource_1
        cv:Action :Action_type-Use_or_Read ;
        cv:Resource :Information_resource-508004-Sensor_data .

    :Role-507710-Expert
        cv:described_in :Business_and_Ogranization_structure-Maintenance_Org_ .

    :Information_resource-508004-Sensor_data
        cv:described_in :Information_space_model-Maintenance_Information_ .
}
```

Code 3: RDF snippet of the access requirement statements

Now the access policies can be specified from the defined access requirements through further performing the steps:

4. Select the resource for which to describe access policies (e.g. *Information resource* in an *Information space model*). The policies describe what actions should be permitted and everything that is not explicitly stated is considered to be denied. This means that by default if no access policies are described nobody can access any resources.

5. Check the necessary access requirements specified for this resource. The access requirements can be determined by either using the functionalities provided by the tool (following references, executing simple queries) or by exposing the models as RDF and executing more sophisticated SPARQL queries like the one shown in Code 2.

6. Specify the access policy for the resource, once the requirements have been gathered. The specific access policy should be specified using the "*Access control requirements*" table of the resource. The table contains four columns:

    a. The first specifies the *Role* that should have access to the resource, covering the "Subject" concept of this access control approach.

b. The second indicates what actions can be performed by the subject, covering the "Action" concept of this access control approach. This works the same was as described in step 3 point a.

c. The third allows describing additional constraints through natural language. Those can address for example things about the "Subject" ("only subjects assigned to the project") or about the general environment ("only during working hours").

d. The forth can be used for *Information resources* to indicate what *Information access* enforces or implements this specific requirement.

The "Resource" portion is represented by the element where the table is described.

Those last steps create the access policy description, which can further be used to implement access control that grants the permissions necessary to execute the described processes. A simple example can be seen in Figure 39 and should be read as: "Expert" (S) should have "Use or Read" (A) access to "Sensor data" (R) [when a Maintenance ticket is created]. However, because "Maintenance Planner" and "Maintenance Technician" are considered to be "Expert", they also should have access to the resource. An example containing the important RDF statements in TriG syntax can be seen in Code 4. Additionally Table 1 provides some examples on how certain access policies can be described using the "*Access control requirements*" table.

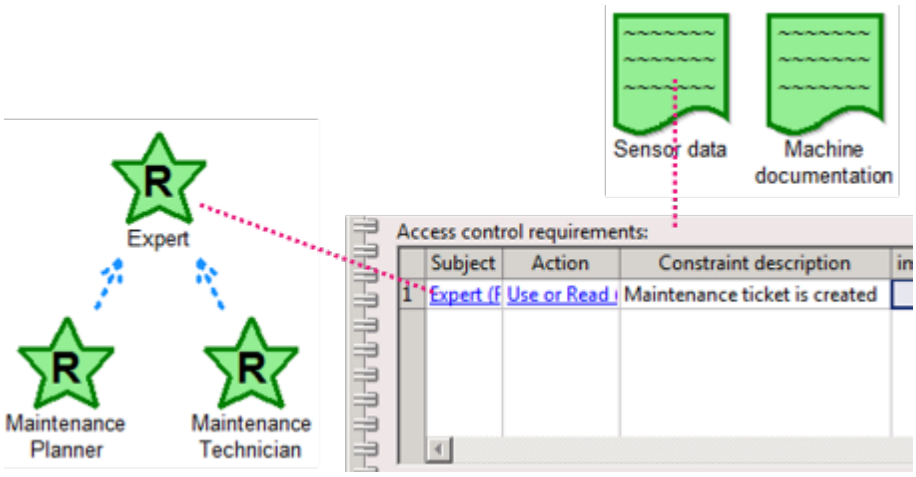

Figure 39: Example showing an access policy description

```
:Information_space_model-Maintenance_Information_ {
    :Information_resource-508004-Sensor_data
        a       cv:Information_resource , cv:Instance_class ;
        cv:Name "Sensor data" ;
        cv:Access_control_requirements
            (:Information_resource-508004-Sensor_data-
Access_control_requirements_1) .

    :Information_resource-508004-Sensor_data-Access_control_requirements_1
        cv:Action :Action_type-Use_or_Read ;
        cv:Constraint_description "Maintenance ticket is created" ;
        cv:Subject :Role-507710-Expert .

    :Role-507710-Expert
        cv:described_in :Business_and_Ogranization_structure-Maintenance_Org_ .
}

:Business_and_Ogranization_structure-Maintenance_Org_ {
    :Role-507710-Expert
        a       cv:Instance_class , cv:Role ;
        cv:Name "Expert" .

    :Role-507704-Maintenance_Planner
        a       cv:Instance_class , cv:Role ;
        cv:Name "Maintenance Planner" ;
        cv:is_a :Role-507710-Expert .

    :Role-507707-Maintenance_Technician
        a       cv:Instance_class , cv:Role ;
        cv:Name "Maintenance Technician" ;
        cv:is_a :Role-507710-Expert .
}
```

Code 4: RDF snippet of the access policy statements

Example 1: Company X operators can see all configuration parameters of machines that they maintain.	
Resource	Machine configuration parameters
Subject	Operator (linked of Company X)
Action	Use or Read
Constraint description	Only for assigned machines
Example 2: Engineers related with the project X can see all the properties of project X, except those properties related with cost.	
Resource	Project properties
Subject	Engineer
Action	Use or Read
Constraint description	Only for assigned project; Except costs; Engineer and Resource related to same project
Example 3: Managers related with the project X can see all the properties of project X, except sensor parameters values.	

Resource	Project properties
Subject	Managers
Action	Use or Read
Constraint description	Only for assigned project; Except sensor values; Manager and Resource related to same project
Example 4: T-Shirt manufacturers can see only the stock quantities, only for T-Shirts that they have sold to Company X.	
Resource	T-Shirt properties (incl. "stock quantity" property)
Subject	Manufacturer
Action	Use or Read
Constraint description	Only T-Shirts sold to Company X; Only stock quantities
Example 4 alternative: Assuming the information resource has a finer granularity - it is not the whole entity, but only its relevant property.	
Resource	T-Shirt stock quantity
Subject	Manufacturer
Action	Use or Read
Constraint description	Only T-Shirts sold to Company X
Example 5: Everyone in Company X can see the amount of time spent by the maintenance operators on machines of their own, but they cannot see which operators have been working on them.	
Resource	Maintenance request properties (incl. "maintenance time" property)
Subject	All (either all Roles, or the highest level Role)
Action	Use or Read
Constraint description	Except operator identities
Example 5 alternative: Assuming the information resource has a finer granularity - it is not the whole entity, but only its relevant property.	
Resource	Maintenance time
Subject	All (either all Roles, or the highest level Role)
Action	Use or Read
Constraint description	---

Table 1: Access control requirement examples

6.2.4 Modelling Mobile Support Requirements

This section describes the necessary steps and guidelines for mobile support requirements modelling. The modelling of mobile support requirements is necessary to establish a better communication between the business expert specifying the requirements for a mobile app, the orchestrator selecting an appropriate app and the developer implementing or adapting the selected app. Among that, the Mobile IT Support model would partly support the developer in the development process. The necessary steps can be divided into two parts:

The first part specifies the steps which are necessary in order to specify the requirements for an app from a business perspective. This part should also allow after it is finished to find existing apps for the specified requirements. Necessary steps in this part are following:

1. Define involved *Roles.* The first step in the modelling of mobile support requirements includes the definition of a *Business and Organization structure* model with several *Roles.*

The defined *Roles* represent required skill-sets (which can further be detailed through their *Skills/Knowledge* table attribute) which should be able to use the created mobile apps.

2. Create a *Process model* in *Business view.* In this step a *Process* model of type *Business view* has to be defined. The *Activities* of this *Process* model have to be connected to the executing Roles that have been previous defined by referencing them in the *"Assigned role"* attribute. An example for a *Process model* and a flat *Business and Organization structure* is displayed in Figure 40.

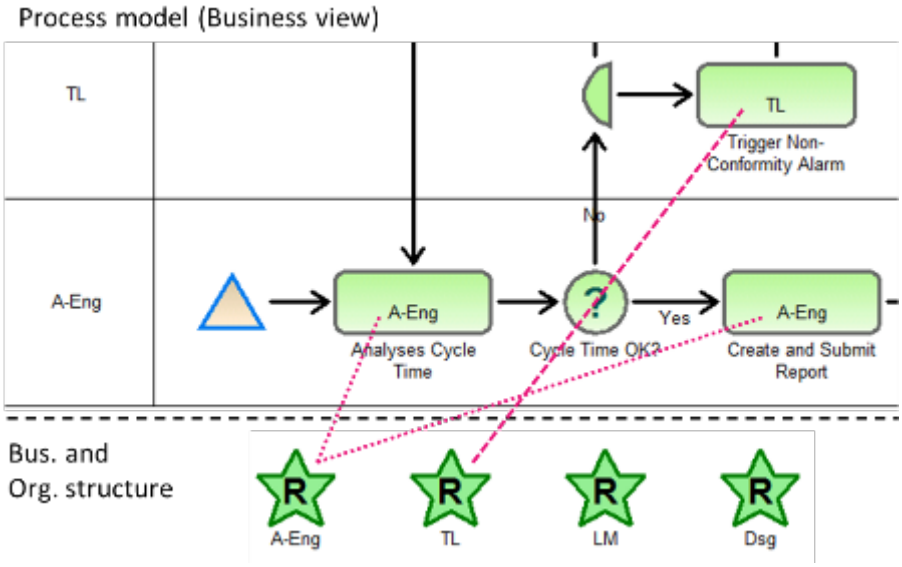

Figure 40: Creating Process model and Business and Organization structure and linking them

3. Create a *Process model* in *App requirements view.* In the next step the *App requirements view* for the previously made *Business view* has to be created. A modelling hint is to copy the *Business view* model and adapt it. Here the activities can be split into several activities? or several activities merged into one as needed to create a proper *App requirements view*. The desired granularity is based on the granularity of the *Mobile IT support features*, such that one *Activity* should have only one *Mobile IT support feature* attached to it. After the *App requirements view* has been finished a *View link* element should be created in the corresponding *Business view* and reference it. An example of the creation of the *App requirements view* and references to the *Roles* is visible in Figure 41. In this example the "Analyse Cycle Time" and the "Create and Submit Report" *Activities* have been split.

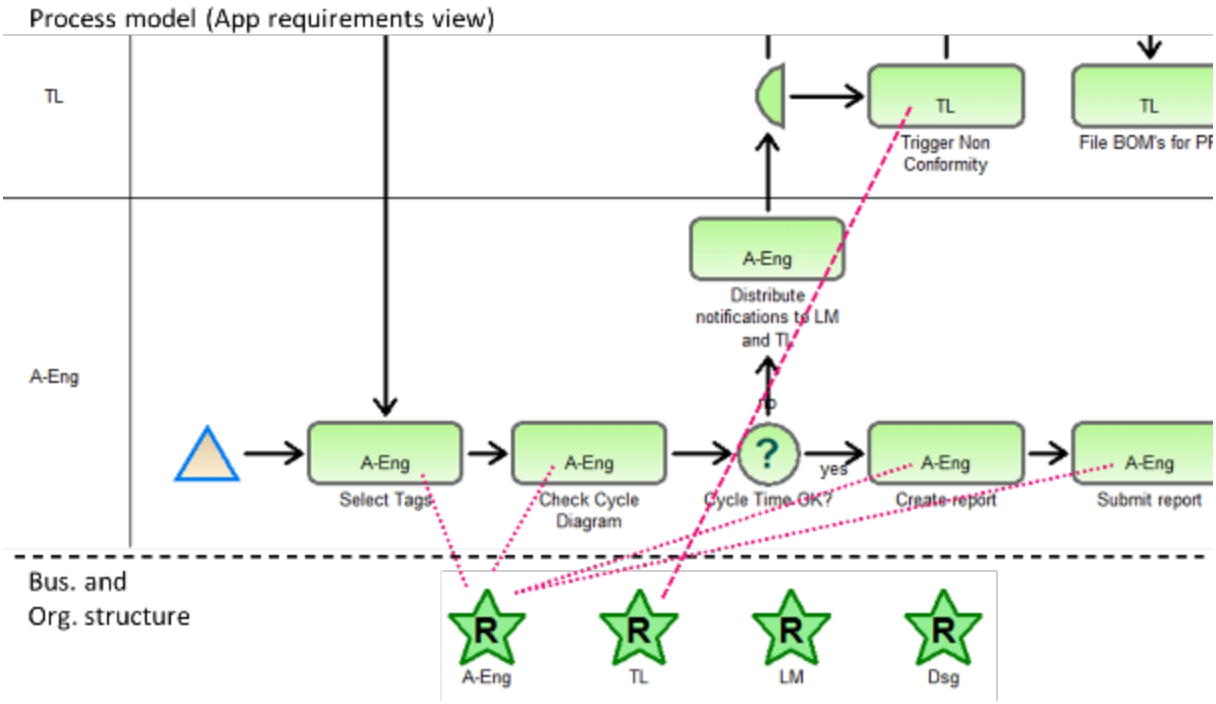

Figure 41: Example of the App requirements view Process model with assigned Roles.

4. Create necessary *Capabilities* and link them to *Activities.* In this step the required *Capabilities* for the app to be developed are defined. The *Capabilities* of the mobile app are defined in a *Resource pool.* Each *Capability* should have a meaningful name which indicates the capability provided by a mobile app (e.g. “Read machine data”, “Change machine state”, etc.). Each of the created *Capabilities* should then be linked to *Activities* of the *App requirements view* which require support by a mobile app. This assignment of *Capability* to *Activity* is done by creating new entries in the “*Required capabilities”* table and referencing the required *Capability.* An example of this step is visible in Figure 42. The defined *Capabilities* can also be used for the discovery of already defined *Mobile IT support features.* The discovery of *Mobile IT support features* can be done either by using the functionalities provided by the tool (following references, executing simple queries) or by exposing the models as RDF and executing more sophisticated SPARQL queries.

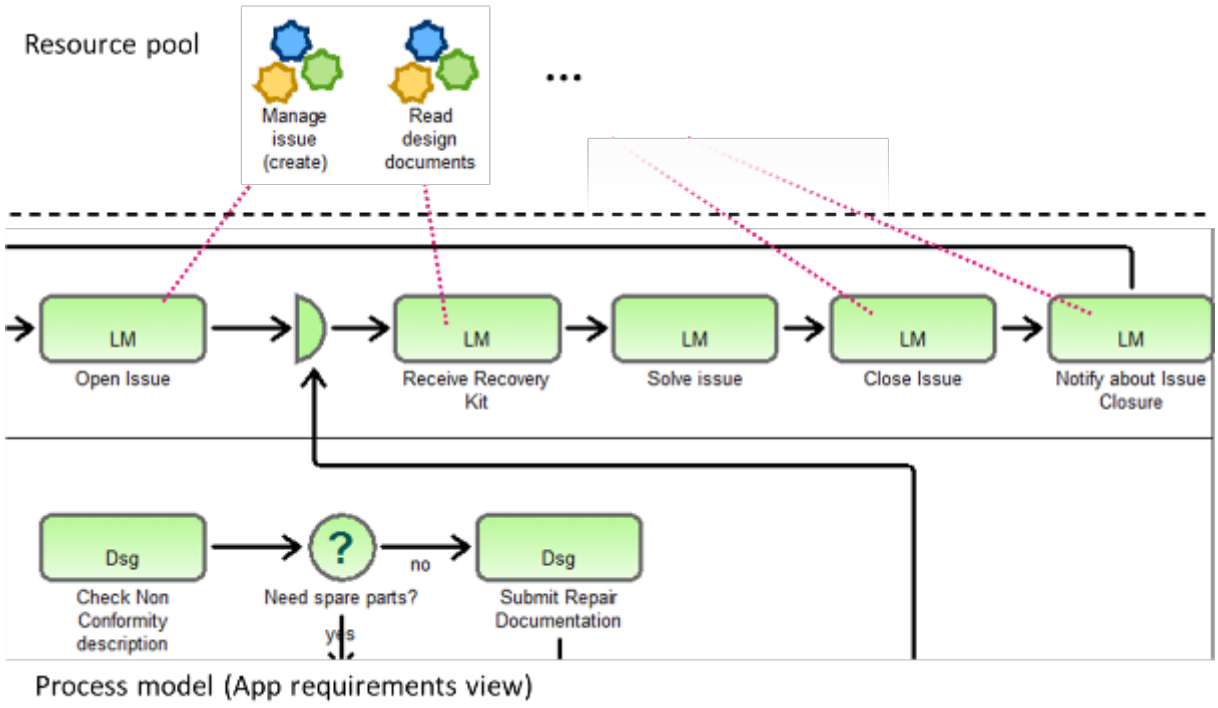

Figure 42: Creating Capabilities and assigning to Activities.

The second part describes a scenario in which no existing apps could be found, and therefore it is necessary to create new apps. This is not necessary if a *Mobile IT support feature* has been found for every *Activity* which needs app support. The following steps are assumed:

5. Define involved *Mobile IT support features* and link them with *Activities.* After the previous step is finished and the *App requirements view* is defined, it is necessary to define *Mobile IT support features* in a *Resource pool.* For this an existing *Resource pool* or a new one can be used. After the *Mobile IT support features* are created in the next step the *Activities* of the *App requirements view* are connected to *Mobile IT support features* if they need app support. An example of the linking between *Activities* and *Mobile IT support features* is shown in Figure 43.

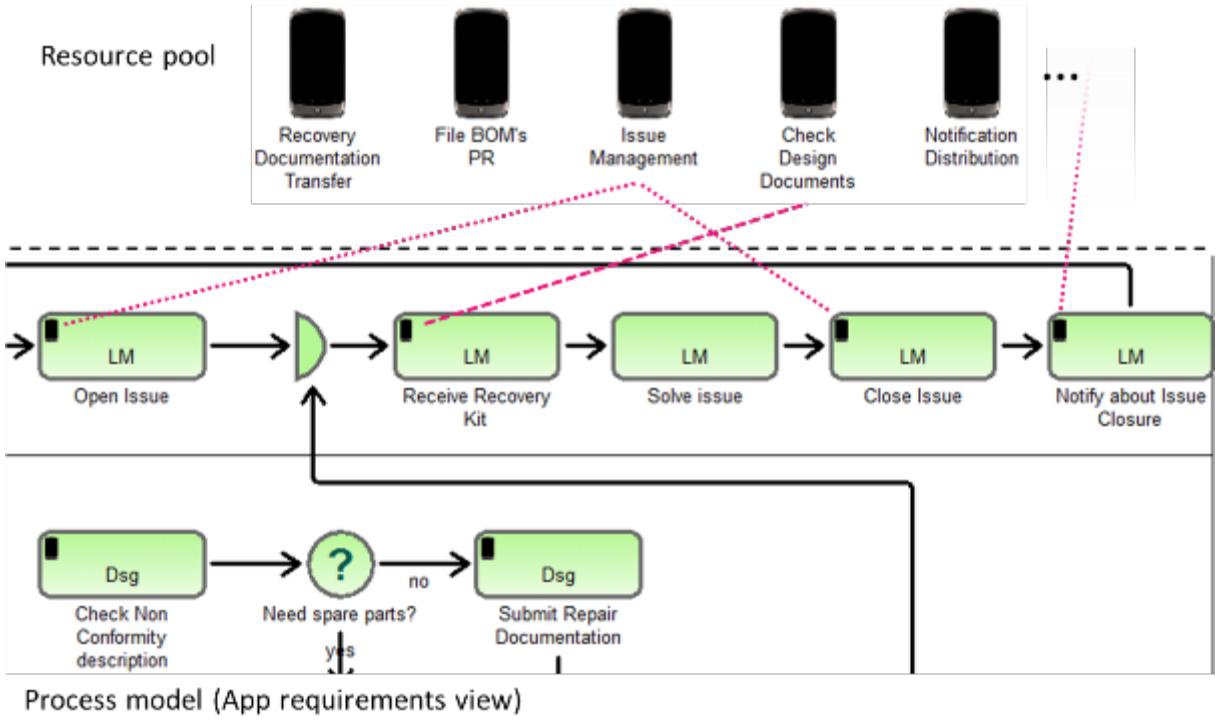

Figure 43: Example of creating Mobile IT support features and linking them with activities.

6. Create *Mobile IT Support model* and link to *Mobile IT Support feature.* In this step the *Mobile IT support* models have to be created by using the initial set of *POIs*. Therefore, it is proposed to copy the objects of each initial set of *POIs* to the newly created *Mobile IT Support* models and start adapting each model. This adaptation includes also the creation of *Screen* objects and arranging the *POIs* inside of it. In order to get a better feeling of the UI the developer can switch the "*Abstraction level"* attribute to *Concrete UI.* After each *Mobile IT Support* model has been created, it is necessary to connect each of those with the corresponding *Mobile IT Support feature* from the *Resource pool*. An example for created *Mobile IT Support* models, the initial sets of *POIs* it was created from and the connection to the *Mobile IT Support features* of the *Resource pool* are visible in Figure 44.

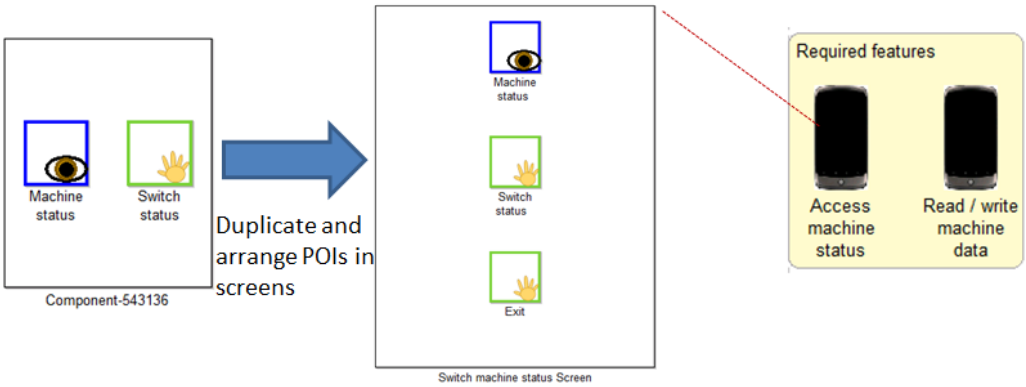

Figure 44: Creation of Mobile IT Support features from initial sets of POIs and connecting the Mobile IT Support models to Mobile IT Support features.

7. Define *Process model* in *Interaction view.* At this step the developer creates possible guidance for each app. This guidance describes the required steps which the user would have to execute and the sequence of appearing *Screens.* The required steps for the user are defined through a *Process* model which is set to *Interaction view.* Each *Activity* of the

Interaction view should be linked to a *POI* of a *Mobile IT Support* model (indicated by a notation switch, in which the icon of the referenced *POI* is visible). Through the linking of the *Interaction view* and the *Mobile IT Support* model it is possible to identify the necessary steps which should be done using a mobile app in order to fulfil a certain task. An example for an *Interaction view* with references to a *Mobile IT Support* model is visible in Figure 45.

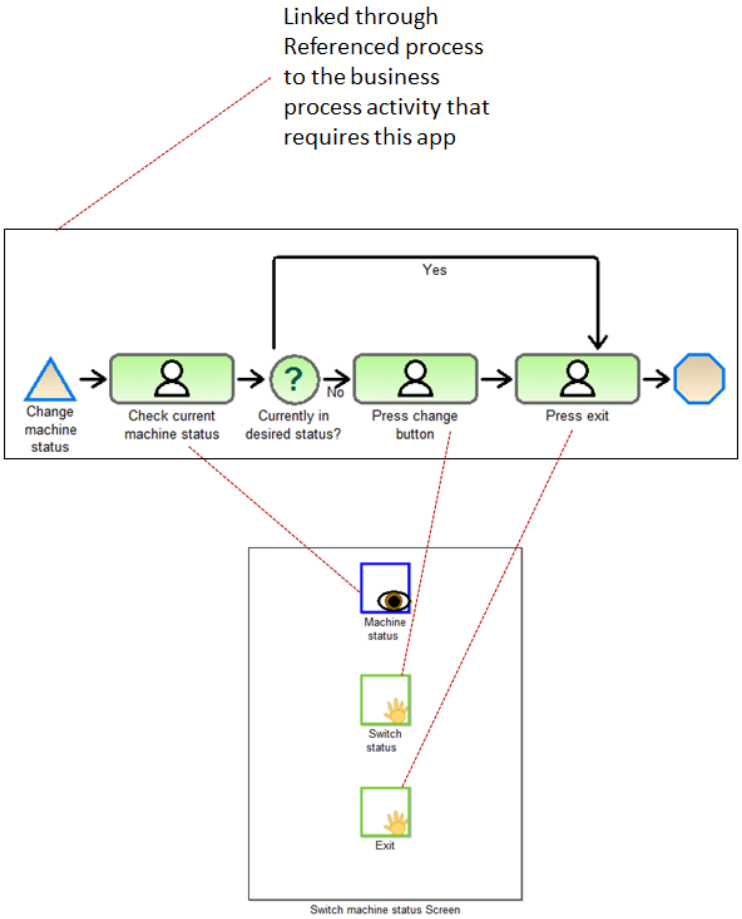

Figure 45: Creation of Interaction view Process model and linking to Mobile IT Support model.

8. Refinement of *Mobile IT Support* and *Interaction view* models to specify final version. After the mentioned *Mobile IT Support* models and *Interaction view Process* model have been created, the next step includes the improvement and refinement of the created models. This step is done by using both models and providing it to the intended user group as test cases. The test cases would verify if the intended user group can follow the required steps and fulfil the given task by using the mobile app. Through the results of the test cases it should be possible to identify various lacks and shortcomings in the model, which could then be improved by the developer. This step should be repeated until the final requirements of the business expert are met and the intended user group is able to use the mobile app to solve the given task.
9. Realisation of design. In the last step the created models would be used by the developer as input for the implementation of the mobile app. This step is done by using different technologies and programming languages for mobile app creation and is executed outside of the modelling toolkit. Therefore, this step will only be partially supported by providing the developer with an RDF export of the created models, which can be further used for app creation.

6.2.5 Modelling Mobile Orchestration

The procedure for describing an orchestration of mobile apps in a model of the current OMI prototype is presented in this section. The goal of the *Orchestration model* is to be used by an orchestration engine. The procedure aims to provide a coherent way of creating the *Orchestration model* starting from the requirements described by the *Process model*. To get from *Process model* to *Orchestration model* perform the following steps:

1. Create a *Process* model in *Business view* and describe what has to be done and by whom (i.e. by which *Role*). In this step the *Business view* that should be supported by the orchestration has to be created and described. It is assumed that the *Business and Organization structure* with all the *Roles* and their hierarchy is already available. If not, then its necessary elements have to be created. The *Roles* are assigned through the "*Assigned role*" attribute of the *Activity*. An example for a properly described *Process model* in *Business view* has been shown in Figure 21.
2. Create corresponding *Process model* in *App requirements view*. This *App requirements view* should be based on the previously created *Business view*. Therefore, it helps to create a copy of the *Business view*, change its view to "App requirements" and start adapting it. Typically activities from the *Business view* are split in the *App requirements view*, but there can be cases where merging of activities might be necessary. The desired granularity is based on the granularity of the *Mobile IT support features*. In the *App requirements view* one *Activity* should have no more than one *Mobile IT support feature* assigned to it. Also link the *App requirements view* to the *Business view* using the *View link* concept.
3. Link *Mobile IT support features* to *Activities* that should be supported. After the basic *App requirements view* has been created it is necessary to link *Mobile IT support features* to the *Activities* that should be supported through the "*Mobile IT support*" attribute. If necessary create new *Mobile IT support features* in the *Resource pool*[17]. Keep in mind that it is not necessary to provide a *Mobile IT support feature* for each *Activity*, only the ones that have to be supported. Also *Mobile IT support features* can and should be reused where applicable. An example for a properly described *Process model* in *App requirements view* with linked *Mobile IT support features* can be seen in Figure 24. It is based on the *Business view* from Figure 21.
4. Create the mobile orchestrations based on the *App requirements view*. Once the *Process model* to be supported has been described in the *App requirements view* it can be used to derive one or several orchestrations. The orchestrations are described as *Orchestration models*. The OMI prototype provides two functionalities to create initial orchestrations based on different assumptions:
    a. The first option creates one orchestration for the whole process, based on the assumption that the whole process will be executed on one mobile device by one role. It can be accessed via item "Create resource orchestration" in the "ComVantage" menu when the "Modelling" component is active.
    b. The second one creates one orchestration for each role participating in the process (determined through the "*Assigned role*" of the *Activities*), based on the assumption that different roles need to collaborate together by using different mobile devices. It can be accessed through the item "Create resource orchestration for each role" in the "ComVantage" menu when the "Modelling" component is active.

    Note that neither one promises that a finished orchestration will be created. Therefore manual adaptations are most likely necessary. Typical adaptations are:

[17] Note that to actually run an orchestration using the *Mobile IT support features* there also needs to be an implementation for them. However, the implementation of apps is not part of the OMI prototype, but section 4.1.2.2.4 describes some steps on how to capture requirements for an app.

a. Remove or change endpoints of created *Followed by* relations.

b. Add transition conditions to *Followed by* relations.

c. Add necessary *Suspension points*, *Path splits* and *Synchronization points*.

d. Add the *Orchestration* element and link it to the corresponding role and process.

Figure 46 shows the automatically generated Orchestration model for the "A-Eng" *Role* for the process from Figure 24 and the previously shown Figure 29 contains the adapted version of the orchestration.

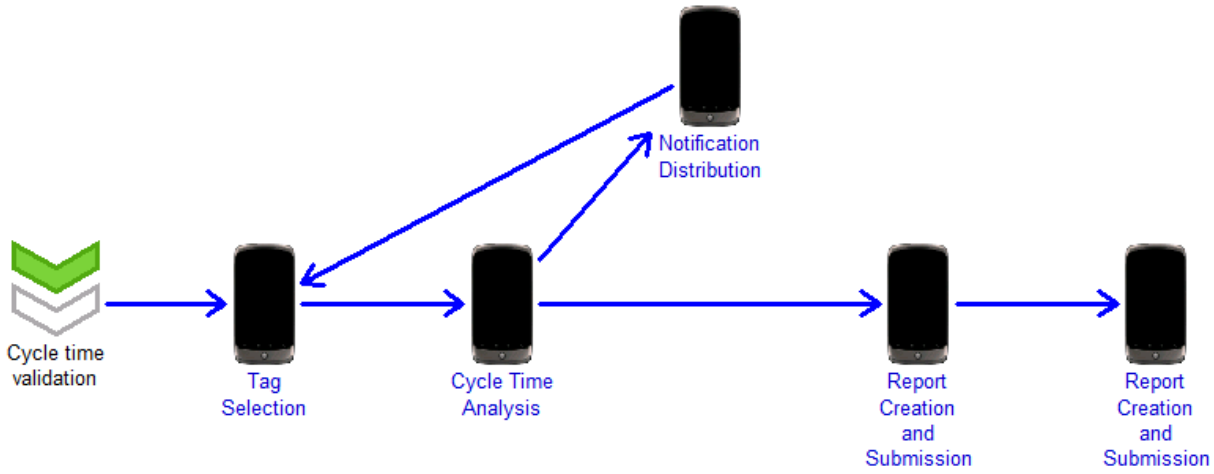

Figure 46: Example for the automatically created orchestration for the A-Eng role from process in Figure 24

After the orchestration models have been created they can be exposed as RDF (see section 1.1.2.2.5) and used by an orchestration engine such as the one developed in WP5.

6.3 Scenario Based Modelling Application Guidelines

This section contains examples focusing on the application areas of the different use case partners. It follows on the envisioned modelling procedure that has been described in D3.1.2 and the partners corresponding adaptation deliverable (D6.2.2, D7.2.2 and D8.2.2). Example models are provided for the scenario specific steps of the procedure and where deemed necessary to further facilitate understanding.

6.3.1 Example inspired by WP6 Scenario

The scenario of WP6 is concerned with the creation and maintenance of a plant that produces cars. As such, a *Value structure* model should be created, which details the services provided to the customers (e.g. plant design, plant building, maintenance etc.). Those can be further decomposed (e.g. maintenance decomposed into checking cycle time) to achieve a useful granularity.

Since the plant is the major value in this scenario, the products it works on (i.e. the car) and the processes employed to create those should be described. **Figure 47** shows an example that describes the car (i.e. the value created at the plant) through a *Value structure*, by decomposing the car into its "Frame", "Engine", "Body" etc. Also the "Body" can be one of three colours ("Black", "White" or "Blue"). The corresponding high-level processes creating the car can be seen in **Figure 48**, describing in what order the car is assembled. The activities of the process should link to the values they work on through the *influences* attribute and they can be further detailed by other processes to achieve a higher granularity. They should also reference the parts that execute them through the *Assigned role* attribute. In this scenario most of the activities are performed by stations or their robots. A lower-level process can be seen in **Figure 49**, which details the process of loading the dashboard and left side member of the car performed at a "Loading station". This station is described in **Figure 50** (and referenced by the activities), which details the parts and most notably the sensors that each of them

has available. The parts of the station are further typed by using the taxonomy described in the *Resource pool* shown in **Figure 51** and the *of type* attribute.

Those descriptions can be further enhanced by adding additional details (e.g. defect descriptions in the *Station structure*, processes detailing how to solve a certain defect etc.) and can be consulted and used to handle or support resolving certain cases (e.g. identifying a defect, optimizing cycle times, retrieving sensor values etc.)

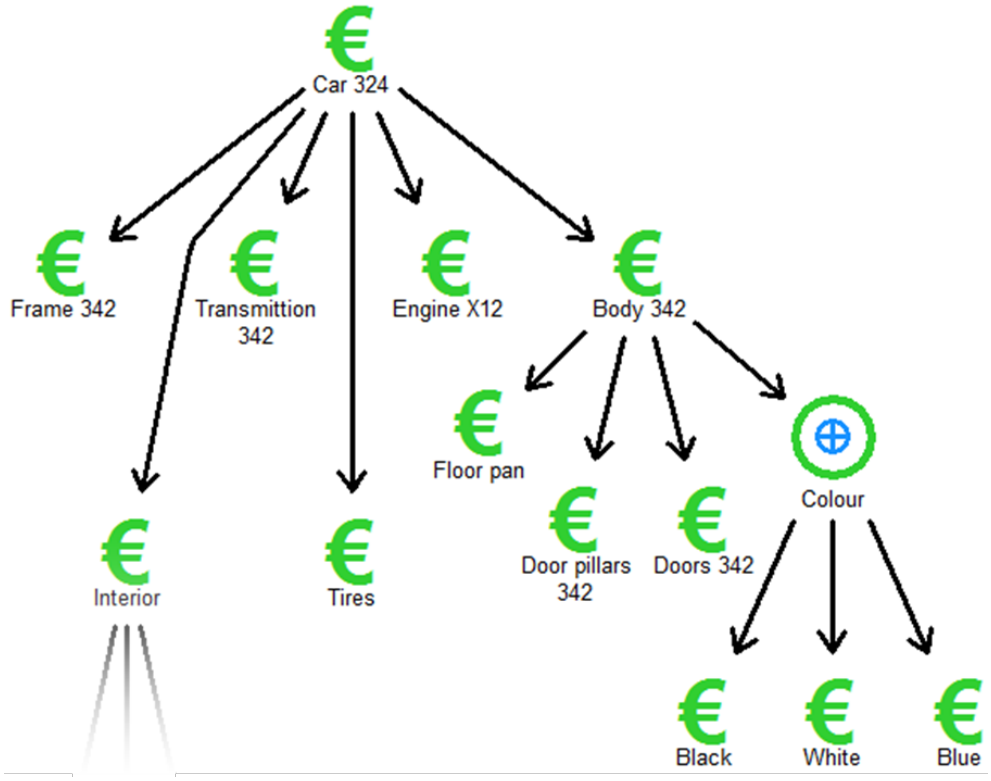

Figure 47 Example Value structure describing a car

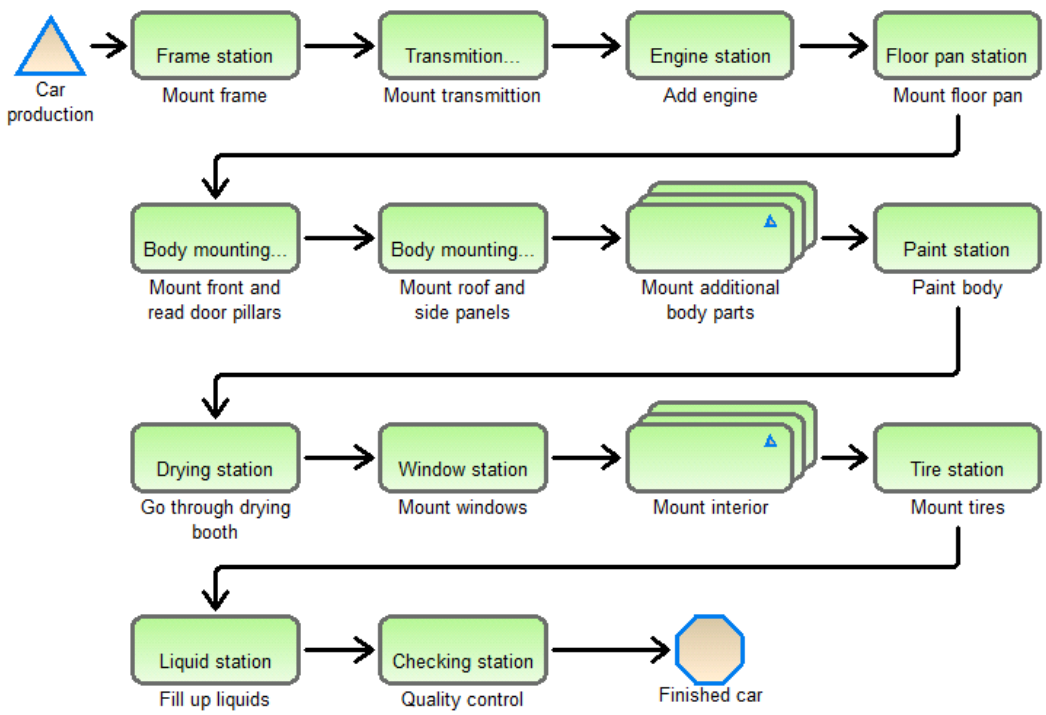

Figure 48 Example high-level Process model describing the car production

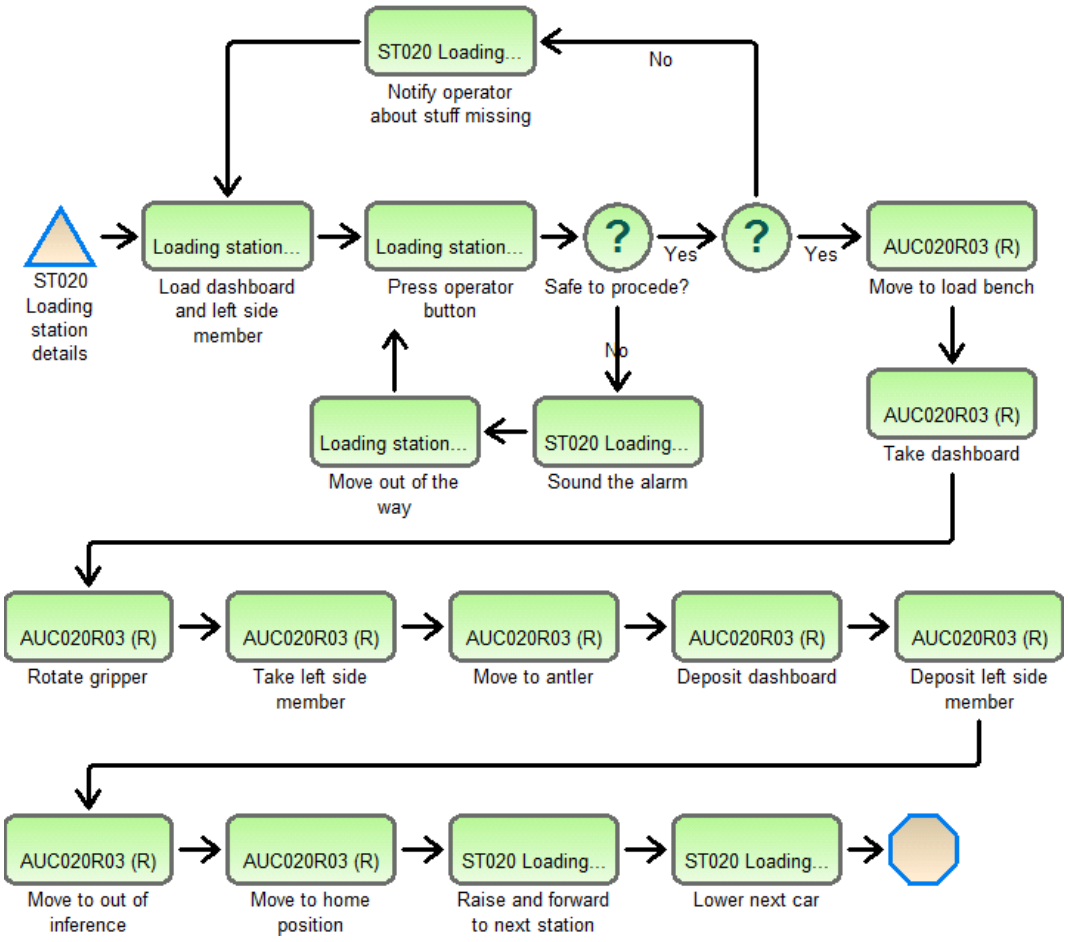

Figure 49 Example low-level Process model detailing the steps performed at a station

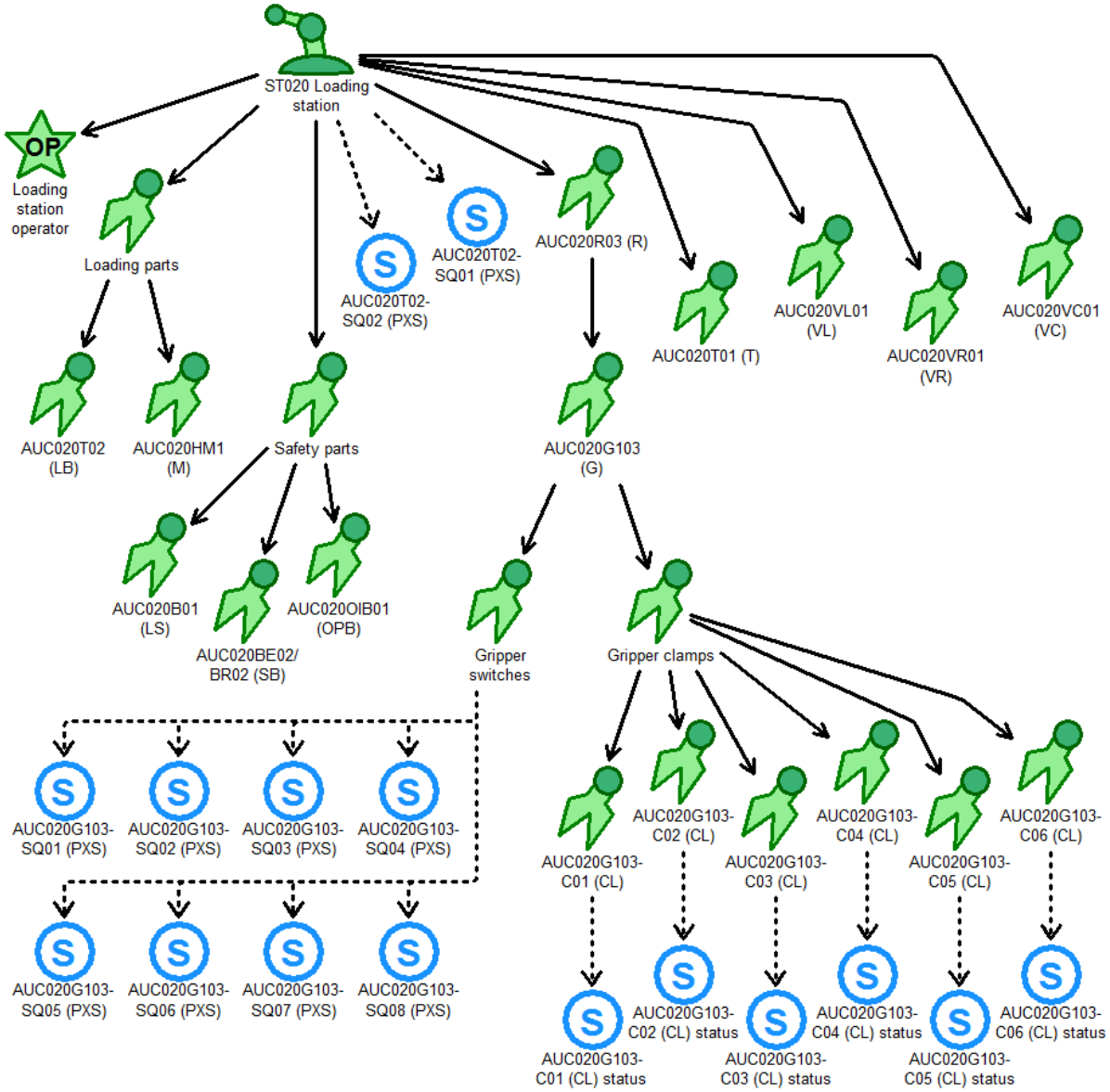

Figure 50 Example Station structure detailing the parts of a loading station

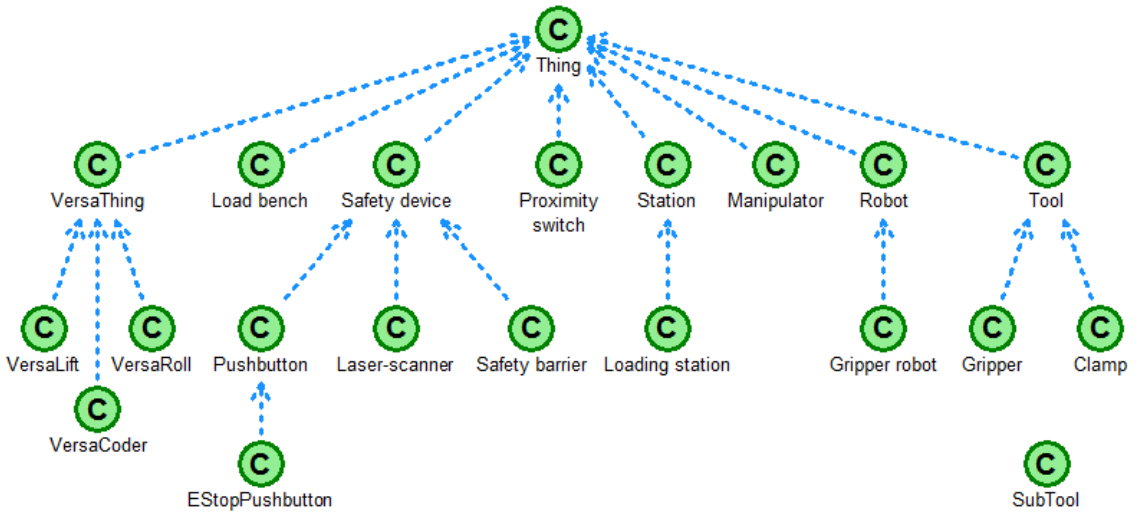

Figure 51 Example Resource pool containing a Component type hierarchy for additional typing of station parts

Afterwards the business processes that are regularly enacted to provide or sustain the previously described values provided by the partner (e.g. plant design, plant building, maintenance etc.) should be described. For this the *Process model* can be used. From there the following steps are no longer application area specific.

The result of an RDF export of these models will enable SPARQL queries such as the example below, which will identify stations or station parts that perform an activity working on a specific value and are of a specific "component type":

```
PREFIX :<http://www.comvantage.eu/example#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX cv:<http://www.comvantage.eu/mm#>
PREFIX owl:<http://www.w3.org/2002/07/owl#>

SELECT ?value ?act ?ass
WHERE {
  # Those bindings represent the input, the first the value to work on
  # The second the component type
  BIND (<http://www.comvantage.eu/example#Value-509000-Engine_X12> AS ?value)
  BIND (<http://www.comvantage.eu/example#Component_type-441759-Loading_station> AS
?ct)

  GRAPH ?proc {
    # Get the activities that influence the value
    ?inf cv:Value ?value .
    ?inflist rdf:rest*/rdf:first ?inf .
    ?act cv:influences ?inflist .
    # Get the thing performing the activity
    ?act cv:Assigned_role ?ass .
    ?ass cv:described_in ?resmod .
  }

  GRAPH ?resmod {
    # Check if the thingis of the right type
    ?ass a ?ct .
  }
}
```

6.3.2 Example inspired by WP7 Scenario

The scenario of WP7 is concerned with the creation and provision of customizable shirts to the customers. For this a *Value structure* should be created that describes the values available to the customer and how they can be customized. Based on this the necessary business partners can be determined and described in a *Business and Organization structure*. From there possible supply chains, which are used to provide the different products, can be designed as *Process models*.

An example for a *Value structure* describing the value decomposition and possible customizations of a shirt can be seen in **Figure 52**, which also contains a specific customization of a shirt (the image with the blue border). Some customization possibilities are the sleeve-length, the shirt colour, a sewing pattern and if the shirt should have a picture embroidered or not. Based on those values a *Business and Organization structure* should be created, which covers *Business roles* and *Business entities* for all the services (i.e. *Values* of type Service) needed to create a shirt. An example can be seen **Figure 53**, where a few *Business entities* for the "Embroidery Shop" *Business role* are present. Using the business structure, the value structure and customization possibilities several high-level processes that handle certain configurations of values should be described to represent the supply chain for

those configurations. The *influences* attribute of the activity should be used to indicate what value(s) it creates and the *Assigned role* attribute should state by whom the activity is performed (e.g. "Embroidery Shop" *Business role*). One example for such a process can be seen in **Figure 54**, which covers configurations that only use the "Embroidery" value, but none of the other (e.g. "Stripe", "Print", "Sewing pattern"). Still this supply chain allows to cover several possible customizations of the shirt, since it does not change for the shirt colour, the sleeve length or the stature.

Alternatively a *Process model* containing several disjointed activities to cover the important services could be created, with *influences* links (through the *influences* table attribute) between the activities and the values. Then, when a product in a certain configuration is ordered, it could be processed and propose activities necessary for the supply chain (based on the *influences* links), which would be modelled at that point.

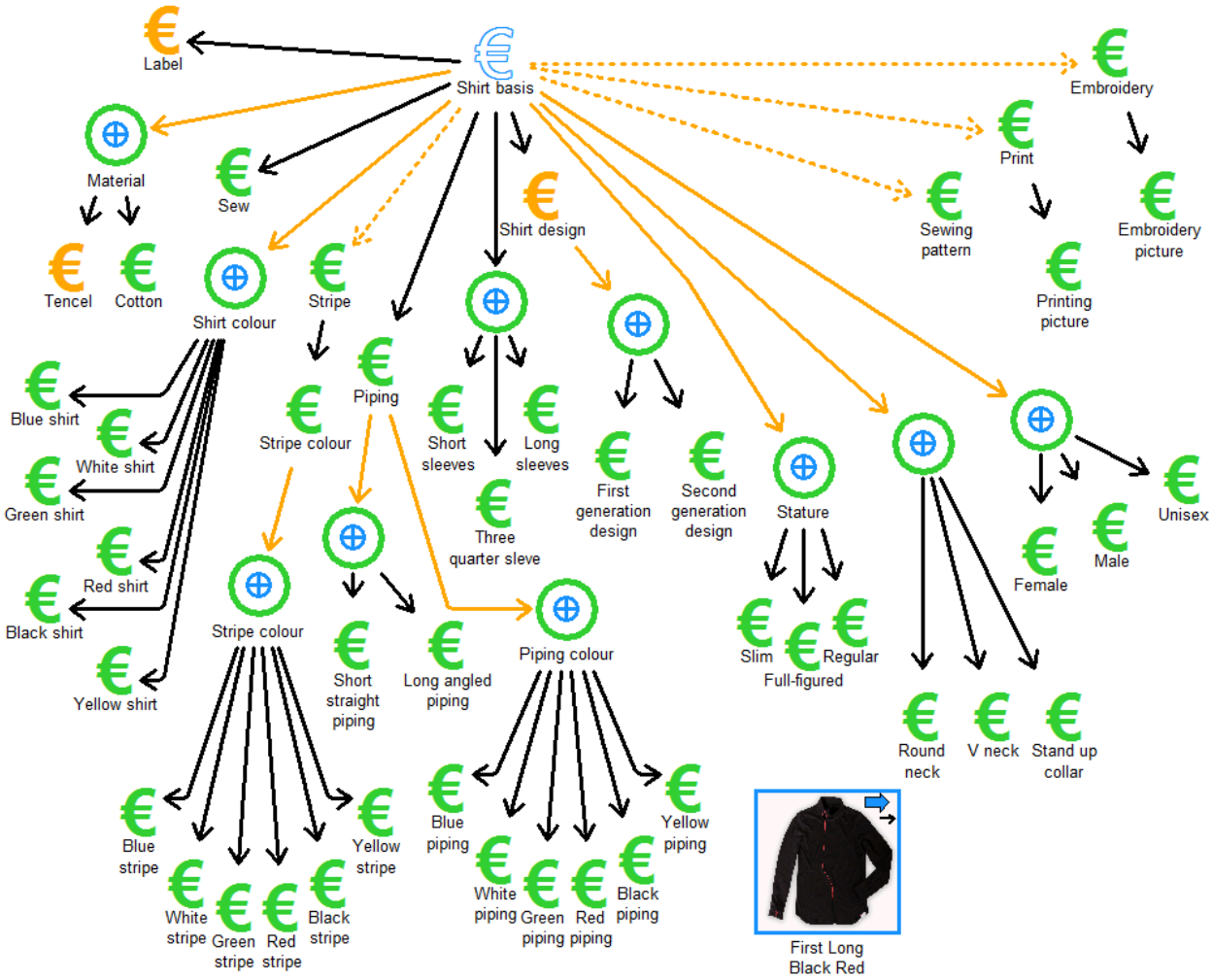

Figure 52 Example Value structure detailing the customization possibilities of a shirt

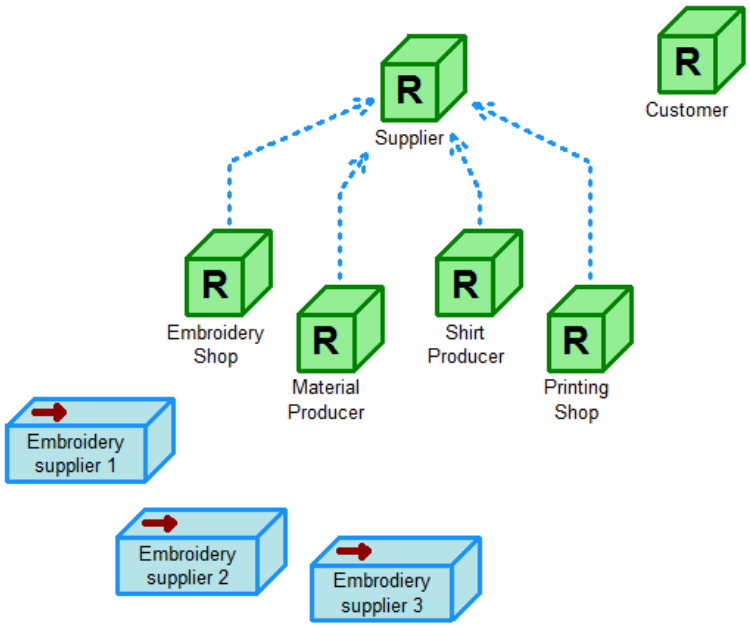

Figure 53 Example Business and Organization structure with necessary Business roles and some Business entities

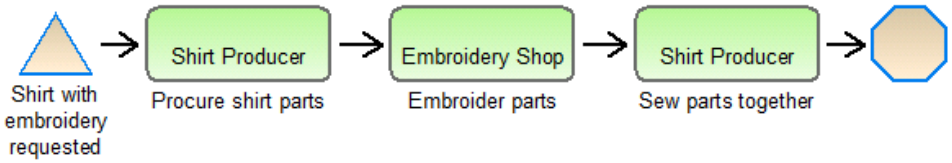

Figure 54 Example high-level Process model for creating a shirt with embroidery (no stripe, print or pattern)

Additionally the processes describing the managerial tasks around the actual production should be described (e.g. customer ordering the product, planning the production, etc.) through a *Process model*.

The result of an RDF export of models will enable SPARQL queries such as the example below, which will identify the activities and performers to participate in the supply chain of a certain value configuration:

```
PREFIX :<http://www.comvantage.eu/example#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX cv:<http://www.comvantage.eu/mm#>
PREFIX owl:<http://www.w3.org/2002/07/owl#>

SELECT ?value ?act ?role ?entity
WHERE {
  # The ?value variable should contain all of the required values from the configuration
  VALUES ?value {
    <http://www.comvantage.eu/example#Value-447250-Sew>
    <http://www.comvantage.eu/example#Value-447260-Embroidery>
    <http://www.comvantage.eu/example#Value-446989-Tencel>
  }

  GRAPH ?proc {
    # Get the activities that influence the value
```

```
    ?inf cv:Value ?value .
    ?inflist rdf:rest*/rdf:first ?inf .
    ?act cv:influences ?inflist .
    # Get the responsible performer
    ?act cv:Assigned_role ?ass .
    ?ass cv:described_in ?busstruct .
  }

  GRAPH ?busstruct {
    # Kind of an if/else-if
    { # If it is a business role, then also look for entities that fulfil it
      ?ass a cv:Business_role .
      BIND (?ass AS ?role)
      OPTIONAL {
        ?entity cv:fulfils ?role .
      }
    } UNION { # Else if it is an entity
      ?ass a cv:Business_entity .
      BIND (?ass AS ?entity)
    }
  }
}
```

6.3.3 Example inspired by WP8 Scenario

The scenario of WP8 is concerned with having maintenance processes mapped on possible machine defects and the (mostly human) resources required to solve such defects. The value provided by a maintenance company are maintenance service contracts, which can be supported by describing contract terms in a *Value structure* as well as the processes performed in order ensure the fulfilment of maintenance contracts (e.g. handling of customer requests, general approach to maintenance, etc.). Furthermore, the *Machine state model* should be used to describe the available machines and detail various known defects that can occur for them.

An example for how SLAs could be described through the *Value structure* is shown in **Figure 55**. Here two SLAs are concerned with maintaining machines ("Standard SLA" and "High-performance SLA") and there is also the possibility to rent the corresponding machines, which include the maintenance SLAs. Also the "High-performance SLA" is a specialization of another value (indicated by the blue arrow and specified by the *configuration of* attribute), in this case the "Standard SLA" by extending it with a faster response time and hotline support. **Figure 56** describes on a high-level how the request of a customer should be handled and certain activities could be further detailed (e.g. "Check request" or "Write report"). The activities have roles assigned (through the *Assigned role* attribute) from the *Business and Organization structure* shown in **Figure 57**. This structure describes the organization of the "Maintenance company" *Business entity*, which has seven *Performers* that fulfil one or several of the shown *Roles* (indicated through the dark red arrow acting as a hyperlink). Also the *Skills/Knowledge* table attribute (of the *Performers* and *Roles*) references the *Skills* as well as *Machine types*, to describe their provided capabilities. One such *Machine type* is detailed in **Figure 58**, which is decomposed into several parts (e.g. "Paper feed", "Printing parts"; transitively also "Fusing unit", "Toner" etc.) that in turn have Sensors and/or Actuators. There are also two Machine defects depicted in the figure ("Empty toner" and "Broken roll"), which can be identified by certain *Sensor* statuses (detailed through the *Defect description* table attribute) and require certain Skills or Knowledge to handle them (detailed through the *Skills/Knowledge* table attribute of the defect). Additionally the "Broken roll" links to the *Process model* (through the *Recommended approach* attribute) shown in **Figure 59** indicate how such a defect should be handled.

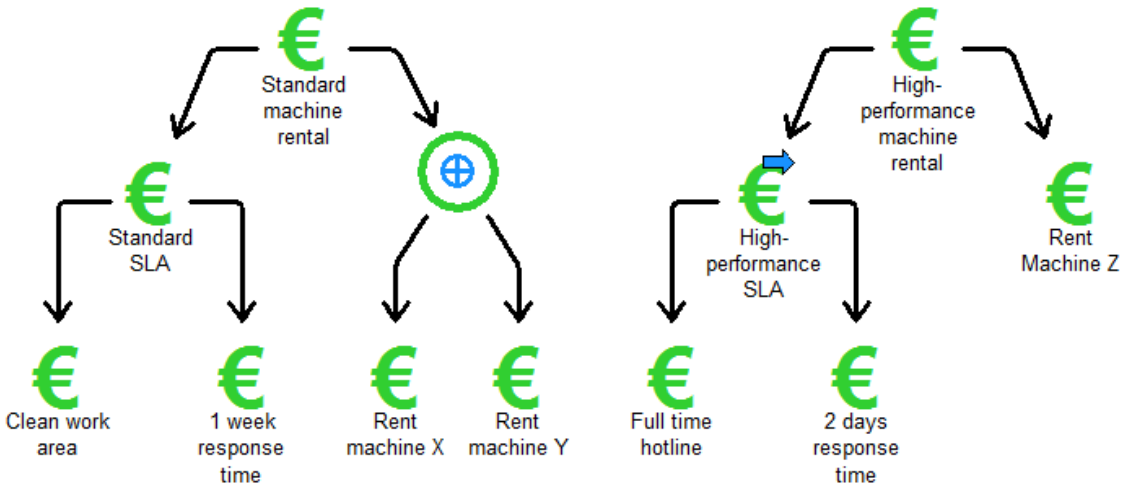

Figure 55 Example Value structure describing SLAs

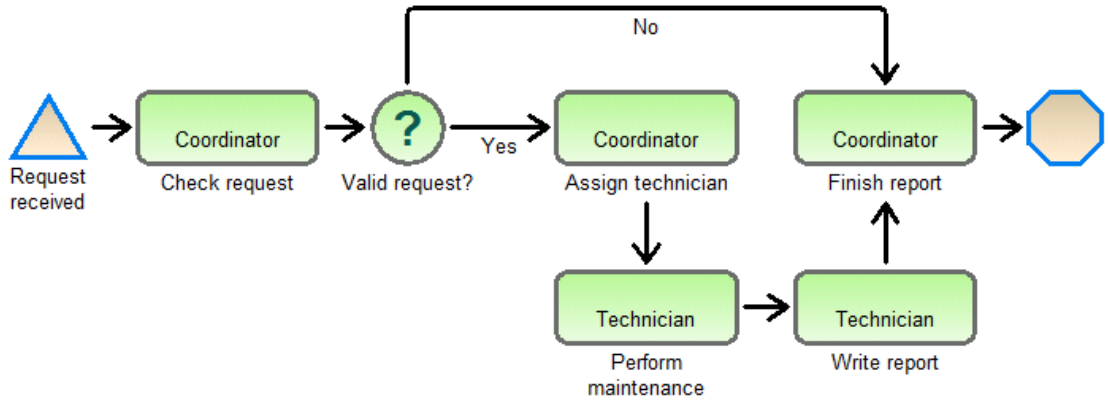

Figure 56 Example Process model for handling a request

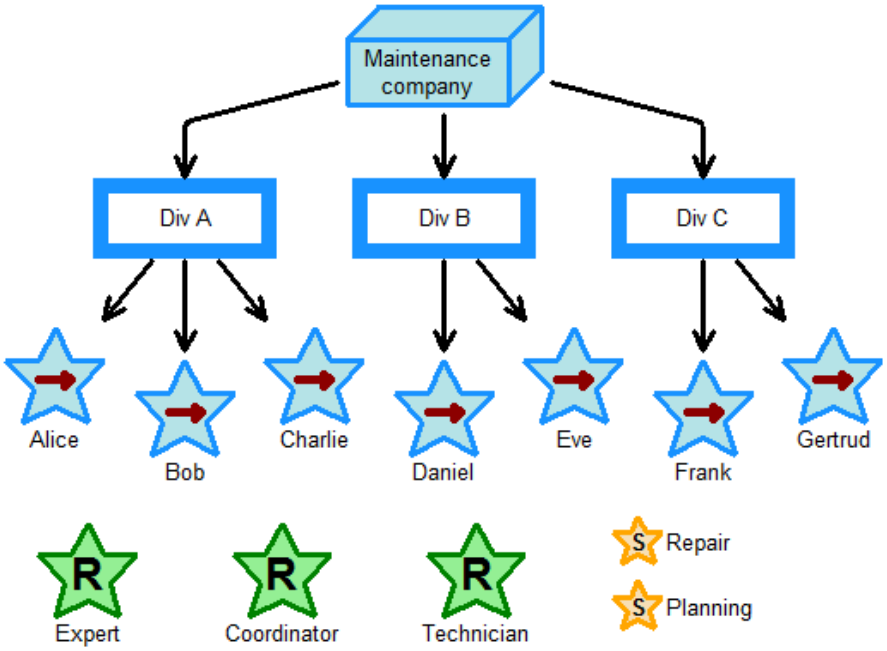

Figure 57 Example Business and Organization structure used by the processes

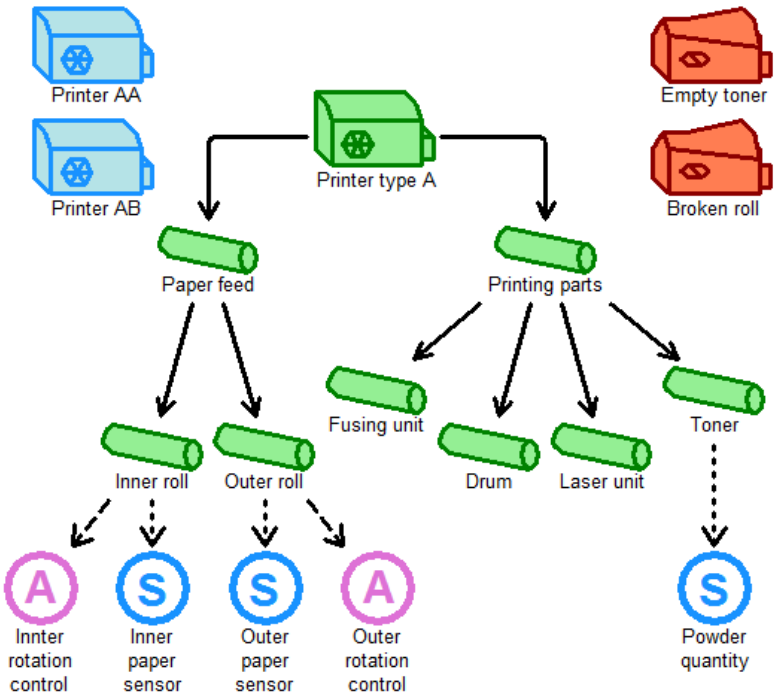

Figure 58 Example Machine state model for a printer

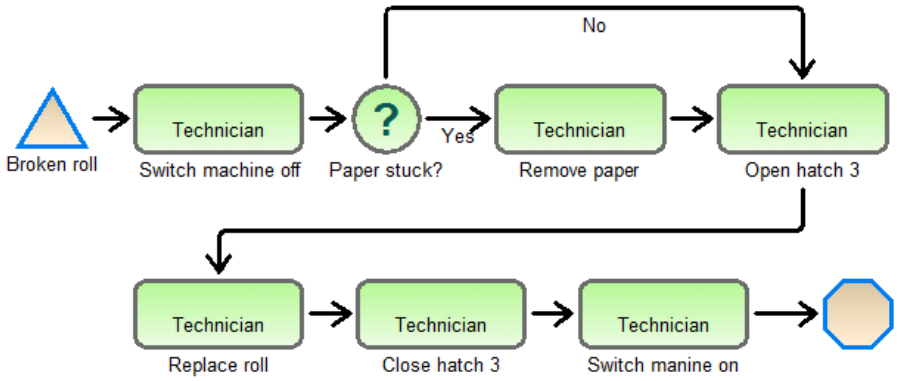

Figure 59 Example Process model detailing how to fix a broken roll

From there the following steps are no longer application area specific.

The result of an RDF export of models will enable SPARQL queries such as the example below, which will identify possible performers that fulfil the skills necessary to solve a defect:

```
PREFIX :<http://www.comvantage.eu/example#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX cv:<http://www.comvantage.eu/mm#>
PREFIX owl:<http://www.w3.org/2002/07/owl#>

SELECT ?mach ?reqCount ?perf (COUNT(?avaSkill) AS ?avaCount)
WHERE {
  # Count how many skills are required by ?mach
  {
    SELECT ?mach (COUNT(?reqSkill) AS ?reqCount)
    WHERE {
      # NOTE # Use this bind to specify for which machine you are looking
```

```
      BIND (<http://www.comvantage.eu/example#Machine_defect-Broken_roll> AS ?mach)
      ?mach cv:SkillsKnowledge ?reqList .
      ?reqList rdf:rest*/rdf:first ?reqSkill .
    }
    GROUP BY ?mach
  }

  # Here find out what specific skills ?mach requires
  ?mach cv:SkillsKnowledge ?reqList .
  ?reqList rdf:rest*/rdf:first ?reqSkill .
  ?reqSkill cv:Element ?reqSkillElem .
  ?reqSkill cv:Level ?reqSkillLev .

  # Here the skills of the performers are determined.
  # Only the ones that fulfil the required skill are considered "available"
  ?perf rdf:type cv:Performer .
  ?perf cv:SkillsKnowledge ?avaList .
  ?avaList rdf:rest*/rdf:first ?avaSkill .
  ?avaSkill cv:Element ?avaSkillElem .
  ?avaSkill cv:Level ?avaSkillLev .
  ?avaSkillElem cv:is_a* ?avaSkillSuperElem .
  FILTER((?avaSkillElem = ?reqSkillElem) || (?avaSkillSuperElem = ?reqSkillElem))
  FILTER(?avaSkillLev >= ?reqSkillLev)
}
GROUP BY ?perf ?mach ?reqCount
HAVING (COUNT(?avaSkill) = ?reqCount)
# Only group where the count of "availalbe" skills is equal to the count of
required
```