Designing for adaptability and evolution in system of systems engineering

DANSE Modelling Formalism, including Domain Metamodel & Semantics: Focused on support for analysis and optimization

D_6.2.3

<table>
<thead>
<tr>
<th>Nature</th>
<th>R</th>
<th>Dissemination Level</th>
<th>PU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project</td>
<td>DANSE</td>
<td>Contract Number</td>
<td>INFSO-ICT-287716</td>
</tr>
<tr>
<td>Deliverable ID</td>
<td>D_6.2.3</td>
<td>Date</td>
<td>31/07/2014</td>
</tr>
<tr>
<td>Status</td>
<td>Final</td>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Contact Person</td>
<td>Christoph Etzien</td>
<td>Organisation</td>
<td>OFFIS e.V.</td>
</tr>
<tr>
<td>Phone</td>
<td></td>
<td>E-Mail</td>
<td><a href="mailto:Christoph.etzien@offis.de">Christoph.etzien@offis.de</a></td>
</tr>
</tbody>
</table>
DANSE Modelling Formalism, including Domain Metamodel & Semantics: Focused on support for analysis and optimization

AUTHORS TABLE

<table>
<thead>
<tr>
<th>Name</th>
<th>Company</th>
<th>E-Mail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christoph Etzien</td>
<td>OFFIS</td>
<td><a href="mailto:christoph.etzien@offis.de">christoph.etzien@offis.de</a></td>
</tr>
<tr>
<td>Tayfun Gezgin</td>
<td>OFFIS</td>
<td><a href="mailto:tayfun.gezgin@offis.de">tayfun.gezgin@offis.de</a></td>
</tr>
<tr>
<td>Roberto Passerone</td>
<td>ALES</td>
<td><a href="mailto:roberto.passerone@ales.eu.com">roberto.passerone@ales.eu.com</a></td>
</tr>
<tr>
<td>Alexandre Arnold</td>
<td>EADS</td>
<td><a href="mailto:alexandre.arnold@eads.net">alexandre.arnold@eads.net</a></td>
</tr>
<tr>
<td>Leonardo Mangeruca</td>
<td>ALES</td>
<td><a href="mailto:Leonardo.mangeruca@ales.eu.com">Leonardo.mangeruca@ales.eu.com</a></td>
</tr>
<tr>
<td>Evgeny Shindin</td>
<td>IBM</td>
<td><a href="mailto:evgensh@il.ibm.com">evgensh@il.ibm.com</a></td>
</tr>
</tbody>
</table>

CHANGE HISTORY

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason for Change</th>
<th>Pages Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>17&lt;sup&gt;th&lt;/sup&gt; July, 2014</td>
<td>Creation of document</td>
<td>All</td>
</tr>
<tr>
<td>0.2</td>
<td>21&lt;sup&gt;st&lt;/sup&gt; July</td>
<td>Structure and content of previous versions</td>
<td>Section 2++</td>
</tr>
<tr>
<td>0.3</td>
<td>28&lt;sup&gt;th&lt;/sup&gt; July</td>
<td>Draft of modelling concepts ready</td>
<td>Section 2.1 / Appendix I</td>
</tr>
<tr>
<td>1.0</td>
<td>31&lt;sup&gt;st&lt;/sup&gt; July</td>
<td>Final version</td>
<td>All</td>
</tr>
</tbody>
</table>
CONTENT

1 INTRODUCTION........................................................................................................................................6

2 SOS MODELING CONCEPTS AND SEMANTICS..................................................................................7
   2.1 MODELING SoS.....................................................................................................................................7
   2.2 DYNAMICITY......................................................................................................................................12
      2.2.1 Graph grammar semantics........................................................................................................14
      2.2.2 Modeling Concepts.....................................................................................................................16
   2.3 CONTRACTS......................................................................................................................................17

3 UNIFIED PROFILE FOR DODAF AND MODAF (UPDM).................................................................22
   3.1 OVERVIEW UPDM............................................................................................................................22
   3.2 RELEVANT SUBSET FOR DANSE....................................................................................................24
   3.3 MAPPING OF CONCEPTS TO UPDM...............................................................................................28

4 EXTENSIONS OF UPDM.........................................................................................................................30
   4.1 MODELING OF STOCHASTIC BEHAVIOUR..................................................................................30
   4.2 CAPTURING TIMING PROPERTIES.................................................................................................31
   4.3 CONCISE MODELING (ARCHITECTURE OPTIMIZATION)................................................................32
      4.3.1 Abstraction layers and mappings...............................................................................................33
      4.3.2 Catalogues..................................................................................................................................34
      4.3.3 Typed connectors.......................................................................................................................35
      4.3.4 Optimization model parameters...............................................................................................35
      4.3.5 Decision variables......................................................................................................................36
      4.3.6 Optimization goals.....................................................................................................................37
      4.3.7 Explicit constraints......................................................................................................................38
      4.3.8 Implicit constraints......................................................................................................................39

5 CONCLUSION...........................................................................................................................................41

6 ABBREVIATIONS AND DEFINITIONS..................................................................................................42

7 REFERENCES.............................................................................................................................................43

8 APPENDIX I: META MODEL..................................................................................................................44
   8.1 COMMON PACKAGE.........................................................................................................................44
      8.1.1 Entity..........................................................................................................................................44
      8.1.2 Unit............................................................................................................................................45

Version | Status  | Date     | Page
1.0      | Final   | 2014-07-31 | 3 of 58
8.1.3 Resource
8.1.4 Information
8.1.5 Event
8.1.6 Matter
8.1.7 Energy
8.2 CORE PACKAGE
8.2.1 ConstituentSystem
8.2.2 Contract
8.2.3 Goal
8.2.4 Infrastructure
8.2.5 ActiveConstituentSystem
8.2.6 SystemOfSystems
8.2.7 Exchange
8.2.8 DynamicityContract
8.3 ADVANCED CONCEPT PACKAGE
8.3.1 Activity
8.3.2 Service
8.3.3 Network
8.3.4 Transition
8.3.5 Rule
8.3.6 Role
8.3.7 Capability
8.3.8 Strategy
8.3.9 Use
8.3.10 Authority
8.3.11 Knowledge
8.3.12 WorldModel
Content of Figures

Figure 2-1: Core Concepts .................................................................................................................... 9
Figure 2-2: Advanced Concepts ........................................................................................................... 11
Figure 2-3: A graph rewriting rule that splits a control node in two .................................................. 13
Figure 2-4: Application of graph rewriting rule to recursively evolve a centralized control structure into a distributed control structure .................................................................................. 13
Figure 2-5: One point evolution .......................................................................................................... 19
Figure 2-6: Guided evolution .............................................................................................................. 19
Figure 2-7: Overlap evolution ............................................................................................................ 20
Figure 3-1: UPDM Viewpoints (Unified Profile for DoDAF and MODAF, January 2012) ................ 22
Figure 3-2: Operational View 2 ......................................................................................................... 25
Figure 3-3: Strategic and Capability View (StV-6) .............................................................................. 26
Figure 3-4: Service Oriented View 5 (SOV-5) .................................................................................. 26
Figure 3-5: System View 1 (SV-1) ..................................................................................................... 27
Figure 3-6: System View SV-10a/SvcV-10a ..................................................................................... 28
Figure 4-1: Left – part of SoS with end to end deadline; Right – changing structure of SoS part .......... 31
Figure 4-2: Packages of a concise model ............................................................................................ 33
Figure 4-3: Example of usage of catalogues ...................................................................................... 34
Figure 4-4: Setting type for typed connector ..................................................................................... 35
Figure 4-5: Usage of auto-calculating parameters ........................................................................... 36
Figure 4-6: Usage of multiply optimization goals ............................................................................. 37
Figure 4-7: Setting tags of optimization goal attribute .................................................................... 38
Figure 4-8: Setting tags for optimization constraint ......................................................................... 39
Figure 4-9: Usage of interface special algebra .................................................................................. 40

Content of Tables

Table 3-1: Mapping of most relevant DMM elements to UPDM .......................................................... 29
1 Introduction

This deliverable summarizes the first two versions and is focused on the support of analysis and optimization.

The first part contains the modelling concepts for Systems of Systems (SoSs) modelling including the semantic concepts. Three different aspects are used to model the SoS specific characteristics of systems. The first are the manifold types of Constituent Systems (CSs) and their (interaction) relations. Beside the technical aspect of interaction also a mean for an intention or goal is required. The second part is the evolutionary development of the SoS and each participating CS. Specifications for behaviour of system dynamics and evolution are addressed as third aspect. The section 2.1 Modeling SoS is completely reworked while section 2.2 Dynamicity and 2.3 Contracts are only updated with minor changes. The second part links those concepts to the UPDM language and the extensions for UPDM as defined in the extension profile (D6.5.2). Section 3.3 Mapping of Concepts to UPDM is completely reworked as well. This document is the harmonization of the first two documents with a focus on the common modelling concepts for modelling SoS.

Section 4 Extensions of UPDM covers also a stable part since the UPDM model did not changed as well as the identified gap. This section is a copy from the previous version of the document (D6.2.2). The presented extensions feed the analysis and optimization while refining and/or extending the SoS model itself. In the appendix the DANSE meta model is documented in detail which has not been done in the previous versions of this document.
2 SoS Modeling Concepts and Semantics

The modeling formalism is threefold. The SoS “Snapshot” covers all participation Constituent Systems (CSs) and relations/interconnections among them. Data and Command exchange between these CS is derived from the behaviors of the individual CSs and their interconnections. The evolution model addresses the evolutionary aspect of SoS i.e. the change of the set of participating CSs or the change of the interconnections among them. This dynamics is called “dynamicy” in the following. The combination of the behavior of the CSs and the dynamicy imply the overall SoS behavior which include e.g. the adaptation to new environmental conditions. To analyze or optimize the SoS one could use all details in the CSs models and the composition of those as SoS snapshot or apply the mean of contract. Contracts restrict the dynamics of the CSs or the SoS on both levels. See (Baumgart, et al., 2011), (Hungar, 2011), (Damm, Hungar, Josko, Peikenkamp, & Stierand, 2011) and (Meyer, 1992) for details how this is applied to the CS behaviors and (Etzien, Gezgin, Fröschle, Henkler, & Rettberg, 2013), (Gezgin, Etzien, Henkler, & Rettberg, 2012) and (Etzien & Gezgin, Correct by Prognosis: Methodology for a Contract-based refinement of Evolution Models, 2014 (to be published)) for how to apply contracts on dynamicy. The DANSE model is defined as SoS = (I, CS, R, D, GC) with I the possibly empty instantiation of the CSs, CS the set of CSs types, R the set of relations/interconnections between CSs, D the dynamic behavior of the SoS (aka Dynamicy), GC the set of goals and contracts of the SoS. The set of constituent systems CS consists of all types of CSs. The instances of cs ∈ CS are connected in the instantiation I with different relations R to each other, to environmental elements e.g. resources and to the contracts and goals in GC.

In the following each of these three concepts are discussed in detail.

2.1 Modeling SoS

The modeling formalism is based upon several concepts which are the building blocks of an ontology. The ontology is used to give an intuitive semantics. Derived from this a meta model is defined which is them mapped to the UPDM meta model.

In (Maier, 1999) five different criteria to distinguish complex from systems of systems are presented. Those criteria refer amongst others to two degrees of freedom (operational and managerial independence) which imply the notion of an agent for a CS.

An Agent represents a CS. This system has a specification or Contract. Agents can be passive or active, which means that they follow their own Goals, or even entire SoS. As illustrated in Figure 2-1, the meta model element “ConstituentSystem” implements the Agent concept with the two sub-types the active “ActiveConstituentSystem” and the passive “Infrastructure”. Ownership is a relation between Agents which implies that if a owns b then b is not fully independent in terms of management from a.
A SoS consists of several participating Agents which themselves might be SoSs. The participation relation is not exclusive. One Agent might be participating in several SoSs at the same time. It also does not necessarily imply the loss of managerial or operational independence of the Agent. Participation is a precondition for emergent\(^1\) behavior and evolutionary development. The creation and/or deletion of Agent in a SoS is directly reflected with the set of participating Agents.

The **Contract** restricts the Behavior observable on the **Interface** of the agent. Contracts use the interface variables and define the valid or invalid behavior over those variables. The **Behavior** of a CS contains all possible system dynamics of the CS. System dynamics are the value assignments per interface variable, per point in time. This (timed) trace might be infinite and characterized via temporal logic (see D6.3.1/D6.3.2 GCSL Specification (DANSE Consortium, 2013)). If an Agent implements its Contracts, it satisfies its Contracts. As long as the analysis did not have proved that the Agent satisfies the Contracts there is a relation **shall satisfy** that indicates which prove obligations are still left. In the meta model this is represented by two lists which contain the associated and the satisfied Contracts.

Each active Agent follows a set of Goals, which is a function over a set of variables and is defined over potentially infinite domains and only the tendency is relevant. A **Goal** is to increase or decrease the value of the function by interaction with the environment (incl. other Agents). The “Contract” element represents the specification of the Agent and “Goal” the objectives of the active Agents.

Within a SoS several Agents exchange different kinds of **Resources** as part of their Behavior. The exchanged item can be **Information, Event, Energy or Matter** (not illustrated in Figure 2-1). These items are provided by one Agent and needed by another Agent. The exchange of Matter or Energy is typically connected with a reduction on the provided side which means if \(a\) exchanges \(x\) with \(b\) then \(a\) is no longer owner of \(x\). Depending on the definition of reduction this is or is not true for Information. One could distinguish exchange and sharing if the exchanged item is not reduced.

---

\(^1\) See D4.2/D4.3 “DANSE Methodology” deliverable (DANSE Consortium, 2013) for definition of emergence
Beside Resources also other elements can be provided or needed. **Capabilities** represent the ability to perform a certain Activity or Service. Agents perform an Activity if the required inputs for the Activity are available and the activity is triggered. In most cases Agents are able to perform several Activities exclusive at the same time and a selection is taken according to what was planned and/or what surfs to feed the Goals. The “Strategy” in Figure 2-2 refers therefore to the performing CS, a planned set of Activities and a set of addressed Goals. Each Strategy combines Activities in order to reach one or several Goals. An Activity changes the environmental state if it is performed. It represents the interaction of an Agent with its environment by defining a subset of its’ Behavior. A Trigger is any Event or condition which initiates an “Activity” or “Service”. Environmental conditions are covered by the meta model element “Knowledge” which is defined individually for each CS as its local view (“WorldModel”). Any sequence or “Network” of Activities, where environmental conditions and Triggers define branches in the execution, is called Service. Agent a provides x of its Capabilities, Services or Resources (x ∈ {Capabilitya ∪ Servicea ∪ Resourcea}).

The Need represents the set of elements required by an Agent. Agent a needs x ∈ (Capability ∪ Service ∪ Resource) if its planned Activity t requires x. If Agent a actually uses the provided elements x_b of another Agent b then

- ∀x_b; b provides x_b, (⇒ element exists)

---

**Figure 2-1: Core Concepts**

2 Colour-Code: White → Abstract Type; Grey → System; Green → Inter CS relations; Orange → Behaviour-related;
• $\forall x_b: a \text{ needs } x_b \ (\rightarrow \text{ element is required})$ and
• $\forall x_b: \exists c \in \{\text{Agents}\}: c \ \text{uses } x_b \ (\rightarrow \text{ element is not used by anyone else})$

In other words: If a resource exists and it is not used by another Agent then it can be used by an Agent.

Any SoS is some kind of a community and in community different general and specific Roles and Rules can be identified. Participation in a SoS requires to assume one of the Roles and to obey a subset of the Rules. A Rule is a restriction of the Behavior of individual Agents or groups of Agents. They are similar to Contracts but are derived from the SoS and applied to the Agents and not part of their specification. Ideally the Rules of a SoS are identical to the specification of the participating Agents but typically Rules are real subsets of the Behavior of the Agents. Even more the Rules typically contradict each other if the associated Role is ignored.

The most important relation between an Agent and a Rule is the obey-relation. An Agent is in principle free to operate according only to its specification and does not need to respect any other regulations but in order to participate in a SoS it has to obey a subset of the Rules of the SoS:

• Agent $a$ obeys Rule $r$ if the Behavior $b_a$ of $r$ is a sub-set of $b_a$ and
• $a$ only performs Activities $t$ which respect Rule $r$.

A Role in the SoS defines what is expected from the Agent if it is participating the SoS. Roles combine the expected behavior in terms of Goals and Rules which mean that behavior is restricted and for active Agents a certain tendency of behavior is required. As an example for Rules and Roles one could think of any communication protocol between machines and/or humans. Participating Agents assume a Role in the SoS which defines the set of Rules they obey and in the case of active Agents a set of Goals they are following. In the case of passive Agents the Role is just a set of Rules to obey. In the other case the tendency of the behavior is typically more important than the set of Rules to obey. E.g. in a negotiation phase the Rules to follow are quite similar for each participant but the position is relevant and decides about the content of the discussion. The assignment of a Role to an Agent can be done by

• The Agent itself or
• Another Agent which has the authority to do so.

In both cases a set of assigned Roles for each Agent is defined. An assigned Role means that the Agent obeys the Rules associated with the Role and follows the Goals associated with the Role.

With authority two different concepts can be meant.

1. To change the Roles or Rules of another Agent and
2. To change the Roles or Rules themselves.

Both aspects are implemented as relations with the same name but defined for different targets. The first authority-relation points to a CS (see “ActiveConstituentSystem”) while the other (see “Authority”) points to a Rule. The first aspect is related to the relation between an employee and his/her line manager. By changing the Role of another Agent, the future behavior of that Agent is strongly impacted. The second aspect is on
the legislation level and may impact the future behavior of all Agents of the SoS. The first authority aspect is strongly connected to \textit{coordination} in general. \textbf{Coordination} means that one Agent $a$ influences the Behavior of another Agent $b$ by

1. Triggering $b$'s next Activities,
2. Command $b$ to perform a certain Activity or to
3. Change $b$'s Role/Rules (Authority)

To \textbf{trigger} Activities of other Agents requires requesting some information according to the \textit{status} of currently executed and planned Activities/Services. With this information (see “WorldModel”/“Knowledge”) one Agent can influence another Agent even if both are on a par / are equal in terms of command hierarchy. In contrast the \textbf{commanding} Agent (see “ActiveConstituentSystem”) is at a higher level in the hierarchy and this allows the Agent to force the other one to perform a certain Activity. This includes also that the commander can cancel currently executed Activities. The highest level of this virtual hierarchy is the authority to change Roles and Rules because this is equivalent to commanding all participating Agents of one SoS. Note that the hierarchy is only to illustrate the different levels of coordination and does not imply the need of any defined hierarchy among the participating Agents.

In this section a rough overview about the meta model and the concepts it implements is given. The entire meta model is contained in the appendix (section 8). The binding brick between the SoS system dynamics and the dynamicity of the SoS is the (self-) reconfiguration which is explained in the following section. The two time scales are conceptually clearly separated but interact with each other. The reconfiguration depends
not only on structural but also on environmental conditions vice versa the system dynamics strongly depends on the number of CSs and how they are connected. From the modeling point of view the reconfiguration must contain conditions only evaluable in the system dynamics time scale and each reconfiguration step is performed without time evaluation in the system dynamics scale. For the specification a certain type of contract, the “DynamicityContract”, is introduced to specify the reconfiguration of the SoS.

### 2.2 Dynamicity

The dynamicity is the structural change of instances of the SoS model. Those changes include changing the relations between CSs and creation/deletion of model elements. Typically these changes affect the participating CSs but could also include the goals/contracts of CSs and the SoS itself. To bridge the gap between the system dynamics, which refers to the internal state of CSs and the exchange items, and dynamicity, which refers to the change of the set of CSs and the relations among them, attributes or rather their values are shared between both aspects. This allows e.g. to model that the fire service is able to buy new fire brigade if its budget is reaching a certain amount. Thereby feedback loops between increasing population and the growth of the fire service can be modelled in a logically correct sense (the opposite direction is symmetric).

The underlying formal semantics was already given in version D6.2.1: Graph rewriting rules (Koenig, 2004) are recipes that turn a graph into another. The idea is to match a pattern graph \( L \) against sub-graphs of the original graph, and replace the matching sub-graph(s) with another graph \( R \). The relation between \( L \) and \( R \) is given by an intermediate graph \( I \) which contains their common elements. More in detail, graph rewriting rules operate on hypergraphs, which extend regular graphs with hyperedges that may connect more than just two nodes. In our context, hyperedges correspond to components, while nodes model their connecting ports and methods. Two components are connected through a port whenever their corresponding hyperedges link to the node corresponding to the port. Multi-party connections are readily represented by this model.

Formally, we can represent a hypergraph \( G \) as a tuple \((V, E, c, l)\), where \( V \) and \( E \) are the sets of nodes and hyperedges; \( c : E \to V^* \) is a connection function that lists the nodes connected by each of the hyperedges; and \( l : E \to \Lambda \) is a labelling function that gives a name to each of the hyperedges. A node labelling function could be used to provide names for the ports, as well.

In order to match graphs, we use morphisms. A hypergraph morphism \( \varphi : G \to G' \) between two graphs \( G \) and \( G' \) is a pair of functions \( \varphi_V \) and \( \varphi_E \) that map nodes and edges of \( G \) to nodes and edges of \( G' \), preserving the connections and the labelling of the edges (i.e., \( \varphi_V(c_G(e)) = c_G'(\varphi_E(e)) \) and \( l_G(e) = l_G'(\varphi_E(e)) \) must hold). A morphism is injective whenever \( \varphi_V \) and \( \varphi_E \) are both injective. In general, we will not distinguish between isomorphic graphs.

A graph rewriting rule \( r \) is a tuple \( r = (L, l, R, \varphi_L, \varphi_R) \) where \( \varphi_L : l \to L \) and \( \varphi_R : l \to R \) are injective graph morphisms, and \( L, l \) and \( R \) are graphs. The idea, as discussed, is to match \( L \) with parts of a graph \( G \), and replace it with \( R \).
A match is formally modelled as an injective morphism $\varphi: L \rightarrow G$. The application of rule $r$ to the match $\varphi$ yields a new graph $H$, such that

$$V_H = (V_G - V_L) \cup V_R \quad \quad E_H = (E_G - E_L) \cup E_R$$

and such that $c_H$ and $l_H$ agree with $c_R$ and $l_R$ on $E_R$ and with $c_G$ and $l_G$ on the remaining edges.

An example of a rule formalized in this way is schematically shown in Figure 2-3. The rule takes a control node connected to at least three other controlled nodes, and splits the control in two to turn a centralized approach into a distributed approach. The morphisms, which are shown only for the hyperedges (the circles) for simplicity, provide the necessary connections between the matching pattern and the resulting graph. In this case, the application of the rule produces the addition of a control node, which is connected to the original node through a link (a node in the graph).

![Figure 2-3: A graph rewriting rule that splits a control node in two](image)

This simple rule can be applied recursively to a graph, in order to split the control. One example of such application is shown in Figure 2-4, where a centralized control structure is first distributed over two control nodes, and finally over three control nodes. The rule is such that the original control node retains the responsibility of communicating with the additional control nodes. Different rules could be devised to also distribute this responsibility, or to construct additional connections between the nodes.

![Figure 2-4: Application of graph rewriting rule to recursively evolve a centralized control structure into a distributed control structure](image)
A graph grammar is simply a set of graph rewriting rules. Applied recursively to a starting graph $G_0$, the graph grammar generates a family of new graphs derived from the initial one. In generic graph grammars, the rules can be (in fact, must be) applied in any order at any time to generate the entire family. In our context, we use graph grammars as methods to specify the evolution of a system, hence we need a way to trigger the application of a rule on a graph. This issue, which was already discussed in deliverable D6.1 "Gap Analysis for existing SoS Modelling Formalisms" (DANSE Consortium, 2012), is solved by adding guards to the rewriting rules that determine when a rule can be applied based on the current state of the system.

### 2.2.1 Graph grammar semantics

Our objective is to give graph grammar rules a semantics which is consistent with the UPDM behavioural models. This will enable us to define precisely the meaning of relations such as satisfaction and compatibility, introduced by the use of contracts, in the context of a dynamically evolving system.

As a starting point, we observe that a UPDM model is composed of various classes of diagrams.

1. **Structural diagrams.** This class of diagrams defines the structure of the model, including the kind of components, or blocks, that may be present together with their attributes, as well as the way these components are connected to build up the system, and the way they are used in typical application scenarios.

2. **Behavioural diagrams.** This class of diagrams specifies the behaviour of components in terms of state-transition systems, dataflow diagrams and/or in terms of message sequence exchanges, ordered in time.

3. **Mapping diagrams.** This class of diagrams provides a link between the functions and behaviours present in the model (specified through the behavioural diagrams) with the blocks and components that actually execute them (specified in the structural diagrams).

Structural diagrams, when taken all together, determine the overall system component interconnections, which can be seen as a hypergraph as described earlier. Behavioural diagrams, on the other hand, specify the actions that must be taken in response to events in the system. At any time, each behavioural diagram expresses the state in which the system resides, determining the kind of actions that can be taken in response to the input. The state is composed of states of state machines, the activation state of functional blocks, and/or the point of execution in message sequence charts. The global state of the system is given by the collection of the individual active states of each behavioural diagram. Because behaviour is tightly linked to components through the mapping diagrams, the behavioural diagrams can also be represented as a graph, which is derived from the structural graph enriched with the behavioural information. Thus, the global state can be represented by a marking of the behavioural graph which, at any time, indicates in which states and in which functions the system is currently executing.

According to this discussion, we may therefore represent a UPDM model as a state machine whose states collect the global state of the system. The state machine is hierarchical and decomposed following the
structure of the system. Transitions are taken on the basis of the interaction between the different components. The collection of the actions performed, and the time at which they are performed, can be used to construct a timed trace of the system, which can be fed to an on-line verification engine or a model checker to verify the properties of interest. This construction, however, is static, i.e., it does not account for the application of the graph rewriting rules so that the interconnection of the components is defined and does not change over the execution of the system.

When a structural change occurs, then we need to change the way components are interconnected, and possibly add or remove components from the system. Because the behavioural graph is derived from the structural graph, we have to change the state machine that represents our system. Our objective, however, would be to have a single model that represents our system across the evolutions, instead of having a number of different models that represent the different stages of evolutions of the system. This is because a single semantics would make it easier to define unambiguously what it means for a property to hold for the entire lifetime of the system.

In order to do that, we follow the technique outlined above and used in Dynamic Input/Output Automata and include the change in structure of the system as state changes of the model. Each state $s$ of the system is composed of two parts: a graph $G$ obtained from the current structure in UPDM, and a marking $M$ of the graph indicating in which state each of the component resides.

Accordingly, we derive the transition relation in two ways:

1. The first follows simply the normal execution of the system, and is consistent with the global hierarchical state machine defined previously. In this case, the transition is between two states that have the same graph $G$, while the marking is updated to reflect the change of states of the components. At the same time, actions are performed and clocks are advanced according to the UPDM specification.

2. The second class of transitions is induced by the graph grammar rules. These are applied to the graph $G$, and are activated by the guard of the rule which rely on the values and the actions performed in the states of the system. The rule produces a new graph $G'$, new attributes and potentially a new marking and actions.

We call the combined (structure, marking) state a configuration of the system. Hence, traditional transitions move the system between configurations with the same structure and different marking, whereas graph rewriting rules give rise to transitions between configurations with different structure, and potentially also different marking (at the very least, new components must be initialized in their initial state, while others may actually transition to another state).

The advantage of this approach is that actions are still part of the state machine, and therefore we can still produce a timed trace, since actions are executed by the system. The timed trace, however, spans the different evolutions of the system, therefore we can analyse the system with respect to the properties expressed as contracts. This provides a precise meaning to the verification problem.
2.2.2 Modeling Concepts

The model of a Systems of Systems or a Constituent System can be interpreted as a graph with labeled edges and labeled nodes. The nodes represent the kinds of model elements and therefore labeled with the (fully qualified) name of the model element. Relations between the model elements, such as the “type” for example, are represented by labeled edges in the graph interpretation. Note that the “type” relation in many modeling languages like UML and its derivatives is a basic relation without further attributes. Relations like “connector” have often further attributes like role names of each end of the connector. Those more complex link-like model artifacts are therefore also nodes in the graph representation because they “carry” additional information. This kind of mapping from the modeling language artifacts to the graph notation allows defining rules for changing the model by translating it into a graph, applying the rule and translating it back to the model. The graph representation is much more generic as the more concrete modeling language and therefore is the semantics of the rules independent form the modeling language. The semantics is independent from the modeling language and its semantics. The challenge is to define rules which do not change the model is such way that its semantics is broken. One could specify a rule which creates elements without any type. This is critical since many modeling languages require an object to have a type. Those constraints depend on the modeling language and any rule should be checked if it violates this constrains.

As we have seen, a rule contains a left-hand side (LHS) and a right-hand side (RHS). In general a rule can be applied to a graph if the LHS is a sub-graph of this graph. The application of the rule changes the sub-graph matched by the LHS to the RHS. Note that all not matched elements of the graph remain. To model a rule at least the following roles of model artifacts for a rule are required:

1. Reader: Elements marked in the rule as reader have to be matched but are not changed. They appear equally in the LHS and RHS.
2. Eraser: LHS elements marked as erasers are removed.
3. Creator: RHS elements marked as creators are added to the graph.
4. Embargo: Since the matching of the LHS defines only the required element one cannot restrict a match without embargos.

For example: If a component in a model is not connected, a rule could add a connection to another component having a port free to connect to. The components are nodes as well as their ports. Between the ports and the components there is an “own” or “containment” relation. A connection is also represented by a node because it has a certain type and maybe additional attributes. The LHS of the rule would require two nodes of type component (or some more detailed type) which must have at least one port each. The match would be restricted to not have a relation from each of the ports to a connector node using the embargo role. The RHS would consist of a new connector node with role creator and with relations to the two ports also in the role creator.

In DANSE, the purpose of graph grammar (the set of graph rewriting rules) is to specify the architectural changes of a SoS due to dynamicity. The focus is on the creation and deletion of CS during the evolution of the SoS as well as on the changing relations between those. Each application of a rule to the SoS model
creates a new SoS model in the sense of a snapshot of the SoS. The goal is to be able to model the possible snapshots of the SoS as a representation of the SoS dynamicity.

2.3 Contracts

The connection between the modeling and the specification means was already presented in D6.2.1 and is repeated in the following:

As discussed previously, a system of systems evolves through dynamic changes in its structure to reach a specific goal, taking advantage of and combining in different ways the capabilities of the constituent systems. The methods that we have described and that we employ to model the dynamical aspect of the system are primarily structural: a graph, representing the interconnection and interaction of the constituent systems, or of parts thereof, is matched against a pattern, possibly mediating through a condition, to result in a new structure that better adapts to a new situation. This may occur over short time scales, for instance in response to an emergency situation, or over longer time frames, such as the adoption of a new technology, or the organic growth of a community and its infrastructure that demand changes in the way these are coordinated.

While the ability to model structural changes is an essential aspect in the description of the evolving nature of a system, the evolution of its requirements is likewise fundamental to properly account for the shifting goals and properties of the system of systems, as well as of the constituent systems. In other words, the specification in terms of contracts and goals must be adapted alongside the system. The events that trigger the adoption of a new specification are the same as those that cause the evolution of the structure, and can therefore be modelled using the same devices described earlier. In this case, different contracts and goals apply at different times. However, we must investigate the semantics of such an evolving specification to clarify how one can go about verifying that the system satisfies or does not satisfy its objective. To do this, we follow the proposal of Zhang et al. (Zhang & Cheng, Using temporal logic to specify adaptive program semantics, October 2006) (Zhang, Goldsby, & Cheng, Modular verification of dynamically adaptive systems, 2009) who developed a temporal logic formalism to specify adaptive program semantics.

The basic approach consists in viewing the system as a composition of a number of steady-state or non-adaptive components, which are able to transition from one another in response to a trigger event. The specification is expressed in terms of goals and contracts, as described in deliverable D6.3.1 (DANSE Consortium, 2013), which uses first order linear temporal logic as its underlying semantics. The objective is to provide a specification of the requirements across an adaptation, i.e., a dynamic change in the system. Temporal logic is in principle able to describe properties over time, and therefore seems suitable for specifying evolving requirements. In particular, the until operator \( U \) appears to be the closest to capturing the meaning of an evolution, since it is able to express the fact that a certain property \( \phi \) must hold until a certain property \( \psi \) holds (\( \phi U \psi \)). However, the semantics of the LTL formula requires that \( \phi \) should hold for a sequence of states \( \sigma \) for all suffixes of \( \sigma \) starting from all the states before a certain state. Instead, one typically requires that \( \phi \) holds only for a certain interval of states in \( \sigma \).
To solve this problem, LTL can be extended with the so called adapt operator (Zhang & Cheng, Using temporal logic to specify adaptive program semantics, October 2006) (Zhang, Goldsby, & Cheng, Modular verification of dynamically adaptive systems, 2009), which we denote as \((\rightarrow \Omega)\). Informally, the semantics of the operator is the following. Assume the system is specified as a steady-state component \(C_1\) which transitions at a certain point in time to another steady-state component \(C_2\), thus modelling the evolution of the system. We say that the system satisfies the requirement \(\phi (\rightarrow \Omega) \psi\) whenever the system initially satisfies \(\phi\) through some behaviour of \(C_1\). Then, when the evolution step takes place, the system stops being constrained by \(\phi\), and starts satisfying \(\psi\) through some behaviour of \(C_2\). The formula \(\Omega\), instead, must be true during the evolution phase, i.e., at the change of state between the two steady-state components.

Formally, LTL formulas are satisfied by sequences of states. We must therefore clarify when a sequence \(\sigma\) satisfies a formula. Assuming \(\sigma\) can be written as \(\sigma = (s_0, s_1, \ldots)\), we say that \(\sigma \models \phi (\rightarrow \Omega) \psi\) if and only if

- there exists a finite subsequence \(\sigma' = (s_0, s_1, \ldots, s_k)\) of \(\sigma\) such that \(\sigma' \models \phi\) when \(\sigma'\) is extended with \(s_k\) to an infinite sequence;
- the sequence \(\sigma'' = (s_{k+1}, s_{k+2}, \ldots)\) satisfies \(\psi\);
- and the sequence \((s_k, s_{k+1})\) satisfies \(\Omega\).

Therefore, the requirements of the system evolve from \(\phi\) to \(\psi\), through \(\Omega\). In simple cases, the property \(\Omega\) can be left out, in which case it simply reduces to the true formula. In other cases, the formula can be used to express conditions that must be true of the system during the evolution process.

Because the system evolves, the requirements (and likewise the behaviours) of a system can be distinguished between those that must hold always, i.e., irrespectively of the evolution of the system, and those instead that change according to the new structure. The former can be specified with the usual specification methods, while the latter require the use of the adaptive form. According to our methodology, requirements, goals and contracts are expressed through a number of patterns by which complex formulas can be constructed (see deliverable D6.3.1). Similarly, we can identify typical evolution patterns that occur commonly, and that precisely define the expected behaviour before the evolution step, the constraints necessary for the evolution, the possible restriction during the evolution process, and whether some degree of overlap can be allowed between the original and the evolved system.

In the following, we summarize three common basic evolution patterns that have been identified in the literature, and that are applicable to our context.

- **One point evolution.** This is the simplest pattern that corresponds to our initial informal example. Under this pattern, the system initially satisfies a specification \(S\), and after the evolution step is triggered by a signal \(A\), it eventually satisfies a specification \(T\). The assumption is that the system reaches a state in which all obligations dictated by \(S\) are fulfilled before it evolves into the new specification \(T\). Including the trigger, the formula that describes this evolution pattern is

\[
(S \land \diamond A) \rightarrow \Omega T
\]
and is visually described in Figure 2-5. As shown, the system transitions to the new configuration some time after receiving the trigger event.

![Figure 2-5: One point evolution](image)

- **Guided evolution.** Unlike the previous case, the system is restricted by a condition R in order for it to reach a safe state in which the evolution step can take place. This could be useful, for instance, when the initial system does not guarantee that some safety requirement is reached in some finite amount of time before a change in the infrastructure can be allowed. The condition R is used to ensure such conditions. In practice, this means that the system goes through two evolution steps: the first is used to reach a state in which it is safe to switch to the new configuration, the second corresponds to the new configuration itself. Formally, this pattern is described by the formula

\[(S \land (\Diamond A \rightarrow \Omega_1 R)) \rightarrow \Omega_2 T\]

Visually, the trace is shown in Figure 2-6. Unlike the previous case, there are now states that must satisfy the additional condition R, as well as the original specification S which remains in effect until the switch is completed.

![Figure 2-6: Guided evolution](image)

- **Overlap evolution.** A more general form of evolution implies the coexistence of both the initial and final specification during the evolution step. This can be useful to account for cases in which the evolution must preserve certain services of the original system for a period of time, while at the same
time provide the new (presumably enhanced) services which are possible under the new configuration. Formally

\[(S \land (\Diamond A \rightarrow \Omega_1 R)) \rightarrow \Omega_2 \text{true} \land (\Diamond A \rightarrow \Omega_1 (T \land (R \rightarrow \Omega_2 \text{true})))\]

meaning that initially the specification S is satisfied by the system. Then, once the evolution step must take place through the event A, the system starts satisfying the specification T as well as a restrictive condition R, all the while still satisfying S. This is expressed by the first evolution step tagged by \(\Omega_1\). In a second evolution step, which takes place at a later time, and once the system has reached a state in which it is safe to abandon the specification S (i.e., all the obligations of S have been satisfied), then only specification T is enforced, and both S and R are dropped. This step is marked by \(\Omega_2\). Pictorially, the situation is shown in Figure 2-7, where the specifications S and T are allowed to overlap to account for a graceful evolution.

![Figure 2-7: Overlap evolution](image)

The same evolving requirement may or may not be satisfied by a system depending on when the system is allowed to transition to the new configuration, and on the state to which the transition is taken. In general, it is safer to restrict the behaviour of the original system before the evolution, so that the right conditions are established in order to reach the new configuration in a safe and known initial condition.

The patterns that we have described above can be composed in order to produce more complex evolution specifications. There are two general forms of composing the patterns.

- **Parallel composition.** In this form, an original system satisfying the specification S may evolve to different alternative specifications \(T_1, T_2, \ldots, T_n\), in response to different triggers. The evolution to be taken may therefore non-deterministically depend on which trigger is received first (e.g., the population of a city grows beyond a certain threshold, or a new technology is introduced to support the communication infrastructure). The resulting specification, in this case, is simply the disjunction of the individual evolution patterns. The specific pattern that is taken depends on the received trigger.
- **Sequential composition.** In this form, the system undergoes a number of different evolutionary steps, taken in a sequence. To achieve this kind of composition, the evolution pattern used to specify the first evolutionary step is used as the original source specification S for the second evolutionary step, and so on. So, using the point evolution pattern, a sequence of two configuration changes from S to T₁ and to T₂ can be formalized as \(((S \land \Diamond A₁) \rightarrow \Omega T₁) \land \Diamond A₂) \rightarrow \Omega T₂\).

By properly combining the composition patterns one can specify complex evolutionary properties using simpler specification blocks.
3 Unified Profile for DoDAF and MODAF (UPDM)

3.1 Overview UPDM

UPDM (Unified Profile for DoDAF and MODAF, January 2012) is a unified Profile for DoDAF (Department of Defence Architecture Framework) and MODAF (Ministry of Defence Architectural Framework). It supports the capabilities to model architectures of complex systems, System of Systems, and service oriented architectures.

Figure 3-1: UPDM Viewpoints (Unified Profile for DoDAF and MODAF, January 2012)

In UPDM model elements are organized in various viewpoints and views. The views and viewpoints which are defined in UPDM are depicted in Figure 3-1. All these views and viewpoints consist of further sub-views such that the contained models are more focussed on certain criteria of the design.

In the following we will give a brief description to all views and viewpoints.

- The **Acquisition and Project Views** (AcV/PV) describe project details, e.g. dependencies between projects. AcV/PV contains the following sub-views: AcV-1/PV-1 (organizational perspective on projects), AcV-2/PV-2 (timeline perspective on projects), and PV-3 (projects realizing capabilities).

- The **Operational View** (OV) is about real-world activities and “answer the “who,” “what,” “when,” “where,” “why,” and “how” of a mission” (Unified Profile for DoDAF and MODAF, January 2012). Contained sub-views: OV-1 (mission or scenario description, list of operational elements), OV-2 (operational node relation description), OV-3 (operational information exchange matrix), OV-4 Actual (relations among resources), OV-4 Typical (organizational structures and interactions), OV-5 (operational activity model), OV-6a (operational rule model), OV-6b (operational state transition...
description), OV-6c (operational event-trace description), and OV-7 (information models on operational architecture).

The **All Views (AVs)** provide an overview description of the considered architecture. Also scope, ownership, and timeframe are represented here. “The AVs include a dictionary of the terms used in the construction of the architecture” (Unified Profile for DoDAF and MODAF, January 2012). AV-1 (overview information), and AV-2 (representation of all elements of architecture as standalone structures).

- The **Strategic Viewpoint (StV)** helps to manage the capability management by providing an overall Enterprise Architecture assessment of the corresponding capabilities and their relationships. In this view, capabilities are introduced, their integration is described, and the realignment or removal are modelled. Contained sub-views: CV-7 (mapping of capabilities and services), StV-1 (strategic context for enterprises), StV-2 (capability taxonomies), StV-3 (capability phasing), StV-4 (dependencies between capabilities), StV-5 (fulfilment of capability requirements), and StV-6 (mapping of capabilities and operational activities).

- The **Systems Viewpoint (SV)** describes realizations of architectures such as resource interaction specifications (SV-1/SvcV-1) or define specifications on functional and non-functional aspects. The models within this viewpoint “represent alternate realizations in terms of equipment capability of the operational capabilities expressed through models in the Operational Viewpoint and in the User Requirements” (Unified Profile for DoDAF and MODAF, January 2012). Contained sub-views: SV-1/SvcV-1 (resource interaction specification), SV-2/SvcV-2 (systems communication description), SV-3/SvcV-3a (resource interaction matrix), SV-4/SvcV-4 (functionality description), SV-5/SvcV-5 (implementation of operational activities), SV-6/SvcV-6 (system data exchange matrix), SV-7/SvcV-7 (resource performance parameters matrix), SV-8/SvcV-8 (change of capability configurations), SV-9/SvcV-9 (technology forecast), SV-10a/SvcV-10a (functional, non-functional specifications), SV-10b/SvcV-10b (resource response description), SV-10c/SvcV-10c (description of interactions between resources), SV-11/DIV-3 (definition of structure of system data), and SV-12 (service provision).

- The **Service-Orientated View (SOV)** is a description of services offered by constituent systems, which are needed to support the operational domain, which are described in the OV. Contained sub-views: SOV-1 (service hierarchy and taxonomy), SOV-2 (service specification), SOV-3 (service mapping view), SOV-4a (service constraints view), SOV-4b (service state model), SOV-4c (service interaction specification), and SOV-5 (service functionality).

- The **Technical Viewpoint** consists of elements describing standards, rules, notations, and conventions “that apply to the implementation of the system architecture” (Unified Profile for DoDAF and MODAF, January 2012). Contained sub-views: TV-1 (technical standards), and TV-2 (technology standard changes).
For more detailed information on the UPDM specification please refer to reference (Unified Profile for DoDAF and MODAF, January 2012). In the following section we will identify the relevant subset of UPDM which is relevant to DANSE.

### 3.2 Relevant Subset for DANSE

The goal of this section is to determine the most relevant elements of UPDM to model (snapshot of) Systems of Systems. For this, we will use the results of section 2, i.e., we will identify those elements of UPDM which realize the needed concepts defined there.

The relevant concepts for modeling identified in section 2.1 are the following:

- Designing and representation of constituent system – We need to design systems individually with independent purposes.
- Services and capabilities – we need to describe the offered services of constituent systems to their environment and their capabilities to realize functionalities.
- Collaboration – we need a concept in order to describe collaboration to reach common goals.
- Dynamicity – we need a concept to capture dynamicity.
- Specification and goals – we need to ensure that systems can work correctly together when changing interconnections and adding or removing systems, and need a concept of defining local and global goals.

In the following we will analyze the domain meta-model of the UPDM 2.0 specification in order to determine all modeling elements, which realize the above needed elements.
Figure 3-2: Operational View 2

In order to capture the first modeling need, i.e. constituent systems with capabilities, UPDM offers “Node” and “SystemResource” as “CapableElement”. For this, consider Figure 3-2, which illustrates the “Operational View 2” which is used for the description of “OperationalNode” relations and the localization of “OperationalCapabilities”. The modeling elements of type “CapableElement” are of ConstituentSystem type in the DANSE meta model (DMM). “Activities” and especially “OperationalActivities” are mapped to the element Activity. The relation between “CapableElement” and “Activities” is illustrated in Figure 3-3 in our model the relation is named “requiredCapability” but addresses the same concept. “Activity” is a parent element of “OperationalActivity” and is related to “Capability” via the relation “MapsToCapability”.

Constituent systems can take on different roles. In UPDM, this is realized by the “NodeRole” element. The inter-constituent system communication is modeled via the “OperationalExchange” relation depicted in Figure 3-2. An “OperationalExchange” is realized between two participating nodes. So we could specify that constituent systems taking on specific roles have to communicate via some protocol. In the DMM the
Exchange element is not refined for a specific operational view and therefore all UPDM "Exchange" relations are equivalent to the DMM Exchange.

Figure 3-3: Strategic and Capability View (StV-6)

Figure 3-4: Service Oriented View 5 (SOV-5)
The above relation of “capable element” and “capability” is only an excerpt of Figure 3-4, which gives this relation in a more direct way. Note, that in DoD services are called “ServiceAccess”, i.e. the diagram SOV-5 maps services to capabilities. Here, the behavior of a service is defined in terms of functions it is expected to perform. In UPDM, services are defined as follows: “A service is described as a unit of work through which a particular Resource provides a useful result to a consuming Resource” (Unified Profile for DoD and MODAF, January 2012). So the UDPM service characterization fits to our needed modelling element of Service.

Figure 3-5: System View 1 (SV-1)

The “System View 1” gives the resource interaction specification. It describes the composition and interaction of resources. “System Resources” are “Capable Elements”, i.e., model elements for Constituent System, as illustrated in the SOV-5 figure. System resources interact via the “ResourceInteraction” relation, which defines the “ResourceInterface”.
In a further system view, i.e. SV-10a, the functional and non-functional constraints on implementations of the considered architecture are specified. So with these elements, we can constrain structural and behavioral elements of the SV viewpoint as resources, functions, data, and ports. *"The constraints are specified in text and may be functional or structural"* (Unified Profile for DoDAF and MODAF, January 2012).

### 3.3 Mapping of Concepts to UPDM

The Table 3-1 lists the mapping between DANSE modeling concepts, UPDM and the DANSE Extension Profile. The core element the **ConstituentSystem** is represented in UPDM in several views. The “System” / “ResourceRole” is the system view representation which allows to model technical relations like “Exchange”. “Own” / “Owner” relations are represented as “Logical-” / “PhysicalArchitecture” indicating the containment of these relations. In the operational view the “OperationalNode” represents the CSs and defines the “operational activities” or “Operations” of a CS. The two subtypes of ConstituentSystem, ActiveConstituentSystem and Infrastructure, are not distinguished in UPDM but the relation to a goal makes them distinguishable. Contracts are mapped to SysML-Requirement and -Constraint.
<table>
<thead>
<tr>
<th>OperationalNode/ Resource</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ActiveConstituentSystem</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>SystemOfSystems</strong></td>
<td>Project</td>
</tr>
<tr>
<td><strong>Infrastructure</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>Contract</strong></td>
<td>Operational-/Resource Constraint, ServicePolicy, Rule, Requirement (SysML)</td>
</tr>
<tr>
<td><strong>Goal</strong></td>
<td>EnterpriseGoal, Mission</td>
</tr>
<tr>
<td><strong>Resource</strong></td>
<td>OperationalExchangeItem, ServiceInterface, ResourceInteractionItem</td>
</tr>
<tr>
<td><strong>Exchange</strong></td>
<td>OperationalExchange, ServiceInteraction, ResourceInteraction</td>
</tr>
<tr>
<td><strong>Use</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Command</strong></td>
<td>Command</td>
</tr>
<tr>
<td><strong>HasAuthorityOver (CS)</strong></td>
<td>Control</td>
</tr>
<tr>
<td><strong>Own / Owner</strong></td>
<td>Relations between ResourcePart and ResourceArtifact</td>
</tr>
<tr>
<td><strong>Activity</strong></td>
<td>Activity</td>
</tr>
<tr>
<td><strong>Rule</strong></td>
<td>Rule</td>
</tr>
<tr>
<td><strong>Role</strong></td>
<td>ResourceRole/ ActualorganizationRole/ NodeRole/ Post/ Organization/</td>
</tr>
<tr>
<td><strong>Capability</strong></td>
<td>Capability</td>
</tr>
</tbody>
</table>

Table 3-1: Mapping of most relevant DMM elements to UPDM
4 Extensions of UPDM

4.1 Modelling of Stochastic Behaviour

For the CAE, a proposal for stochastic modelling has been worked out and is presented as an overview in this subsection. It is based on a set of attribute stereotypes that can be applied to any block attribute. This idea is close to the suggestion of the non-normative distribution extensions made in appendix of the SysML 1.3 specification, but adds the possibility to regenerate a distribution-based random value whenever needed (and not only at initialization). Being able to regenerate a random number based on a same distribution enables to model real-world behaviours, such as the time needed for a human to perform a task, which is repeated over time.

The chosen implementation in the CAE Rhapsody model was driven by the following requirements:

- be simple enough for the end-user (the SoS architect)
- be easily readable and exploitable by the run-time technologies
- be consistent and make sense from a modelling point of view
- be able to use a single distribution to generate several random values

Stochastic behaviour is classically represented by the concept of random variable. The probability distribution of the variable is interpreted as the probability that the variable takes up a certain value when it is observed. To follow this classical approach, given a random variable R that takes values over the reals and a real variable V, the assignment “V = R_observe()” can be seen as an observation of the random variable, provided that “R_observe()” function is an automatically or manually defined to generate new random values.

In order to include stochastic aspects into the UPDM/SysML model, following stereotypes are proposed for the DANSE profile:

1. Numbers (Real and Integer) with a uniform distribution → min and max properties
2. Numbers with a normal distribution → mean and standard deviation properties
3. Numbers with a custom distribution → custom ‘observe’ function property (user-defined)

These three different kinds of generic random variables are a compromise between:

- Usability → pick one variable kind from a set of predefined ones (library)
- Expressiveness → enable the user to specify custom ones; furthermore, by allowing all the above properties to be defined as ‘String’, the random variables could support expressions which refer to the model, which especially enables to define causal dependencies to the state of the model (e.g. take time into account)

Please refer to the CAE deliverable to see application examples of these concepts.

Another stochastic aspect that is still to be considered for the modelling formalism is the ability to express the probability that one transition is taken rather than another one in the behavioural model. The same comment applies at the dynamicity level, where the creation/deletion of constituent systems or change of relations between them should be stochastically quantifiable.
4.2 Capturing Timing Properties

Timing is a crucial issue in safety critical scenarios at both system and SoS level as the correct functionality directly depends on the timely operation of all interacting parts of a considered part of a SoS or system. The execution of services is related to time as these naturally cannot be executed instantaneously. For this, we need timed behaviour models such as timed automata or timed sequence charts. Due to complexity, it could be desirable to have black box views to constituent systems, where the exact execution semantics is abstracted and only relevant properties are kept. For this, contracts are a suitable formalism to capture such properties. As complex interdependencies between several services executed on a single resource or system could lead to delays in the executions, it is not obvious whether these contracts are always fulfilled. As other parts of the SoS depend on these timing contracts, it is a crucial verification task whether all constituent systems and parts of the SoS adhere to their contracts.

End to end deadlines determine the allowed execution time which is allowed to pass from the triggering of the corresponding part of the SoS to its response. If such timing issues are violated, the correct functionality of the system or SoS will be affected. Consider for example Figure 4-1. The considered part of a SoS is decomposed into further constituent systems on which a set of services are allocated. The part of the considered SoS is annotated by a contract consisting of an environmental assumption (A) and a guarantee (G). It is required, that the delay between trigger ‘a’ and response ‘g’ is between \([t_1, t_2]\). For this, the SoS part assumes that ‘a’ occurs with a particular period. As a set of services are allocated on each constituent system which cannot be executed in parallel, it is not obvious whether this decomposition structure satisfies the requirement.

Figure 4-1: Left – part of SoS with end to end deadline; Right – changing structure of SoS part.

Further we have to take into account that changes of the structure may occur. For this, consider the right part of Figure 4-1. For the new decomposition structure of the part of the SoS it has to be guaranteed whether the contract still holds.
In order to have correctly interacting constituent systems and systems working in a timely manner, both systems and parts of the SoS need to be annotated by timing contracts. Further, these timing specifications need to be captured in a rigorous way in order to enable automatic verification. So far, these specifications cannot be captured by UPDM.

4.3 Concise Modeling (Architecture Optimization)

Concise Modeling is developed in order to solve several main problems in large-scale system prototyping:

- It is aiding the modeler to have fewer elements in a model that is being created manually in the tool (concrete model), than the actual model that is defined by it (the "expanded" model). The expanded model will usually have more instances (parts) than the concise model; it will also have realizations of all the links that are defined in a concise model.

- Reducing the number of elements that are in the model we create is very important, because a vast amount of elements to take into account in large systems makes the model cluttered, reducing possibility of concentration on important aspects. Thus, a concise model makes it easier to understand, to modify and update, and to verify the model.

- It uses a database structure in order to supplement the concise model information. It means that the whole "picture" of the expanded model is composed of the information in the concise model and in the database (It is also dependent on the decisions of the optimization engine, but it is described in the next point). It is helpful in two ways:
  - First, we can have one element in a concise model that will be "populated" from a database list (It can spare adding a lot of information that is important for the model, but unimportant from the modeling point of view).
  - Second, we can have one block in a concise model that will be "realized" with one of the alternatives we have in our database (It is also related to optimization that is described next).

- The concise plug-in processes the concise model and the database information to create an input for an optimization engine. An optimization engine then performs design space exploration according to the criteria appearing in the concise model.
  - Using all the information from the concise model and the database, the plug-in and an optimization engine create a design space (meaning, all the possible variants of the expanded models, confirming to the criteria arising from the concise model). Everything in the model influences the design space - the blocks and their multiplicities, the connections and multiplicities at their ends, the mappings between levels of abstraction and the constraints that are attached to the blocks or to the attributes in the model (constraints are written in an OPL language).
This way we get not only a compact and clear way to model a difficult and large-scale system, but we also have an optimization program to perform design space exploration and propose us the solutions on the efficiency frontier of the design space. Moreover, concise modeling is designed to support the multi-criteria (multi-objective) optimization. Multi-criteria optimization is very important in real-world designs, where we want to know the values of different optimization objectives (such as cost, weight, etc) individually for each design alternative, rather than some single aggregated value. Multi-objective optimization provides more precise and clear information to the decision maker, than a single-objective optimization. As we want to add more metrics (such as reliability, complexity, etc), the advantage of the multi-objective optimization approach becomes more prominent.

The full list of stereotypes and tags using for concise modeling can be found in the D6.5.2 Extension of standard profiles for DANSE Modeling (DANSE Consortium, 2014).

### 4.3.1 Abstraction layers and mappings

The concise modeling approach has three layers in three different model packages as represented in Figure 4-2: Packages of a concise model.

- **Functional layer** – serves as the requirements definition for the system architecture.
  - May be modeled concisely in some cases, but all parts and links will be explicit
  - May be a result of a higher abstraction iteration using the same approach
  - May have connecting links
  All parts and links of functional layer must have <<functional>> stereotype.
- **Technical layer** – architecture modeling plane. Modeling is based on the requirements of the functional layer. The objects on this plane usually represent real components (or subcomponents) and real flows between them (data, energy). All parts and links of technical layer must have <<technical>> stereotype.
• Geometrical layer – used to index the objects of the technical layer. Sometimes this layer directly represents the geometry of the system and is used as such. For example the instances of this layer may represent possible placeholders for the actual components on the technical plane with the optimization process tasked with finding the right combination of components and their locations. Alternatively, this layer can be an abstract collection of indices bounded by constraints. All parts and links of geometrical layer must have <<geometrical>> stereotype.

• Mapping – the way to relate one layer to the other. Mapping is done by using the SysML «allocate» dependency. An object on the functional plane can only be mapped to one object on the technical plane, as otherwise there would be ambiguity in the definition. However, any number of objects on the functional plane can be mapped to a single object on the technical plane. If a multiple mapping is indicated, the meaning is that the optimization must select the best mapping subject to constraints and rules. Mappings from functional to technical layer must have <<mappedTo>> stereotype. Mappings from technical to geometrical layer must have <<allocateTo>> stereotype.

4.3.2 Catalogues

A technical part in the concise model represent some physical element (or number of physical elements) of the some specific type. In general case the actual parameters of this physical element (or these physical elements) are not known during modelling process, but must be chosen from some catalogue by optimization solver. I.e., simplifying the actual process, we can say that optimization solver "picks" most suitable physical element in place of technical part in the concise model from the catalogue of different (having different attributes) physical elements of specific type. To achieve this behaviour in the concise model a <<catalog>> stereotype must be applied to the technical block from which corresponding technical part instantiated. The <<catalog>> stereotype also must be applied to all attributes of this block which values are chosen from the catalogue. For example, if we model antennas for LTE network and name, cost and antennalId parameters must be chosen from catalogue of different LTE antennas, then we must create block with corresponding attributes and apply <<catalog>> stereotype to this block and to the corresponding attributes as shown on Figure 4-3: Example of usage of catalogues.

Figure 4-3: Example of usage of catalogues
4.3.3 Typed connectors

Connectors between technical parts of the system model are often represents various physical elements such as cables, shafts, ducts, pipes, etc. The typed connectors used to bring this relationship into concise model. Typed connector is a connector that has some technical part behind it. This technical part is of type (of block) which represent some class of physical elements serving as connectors e.g. power cables, water pipes, etc. Applying stereotype <<TypedConnector>> to a connector causes a tag “type” to appear (to be added) among the properties of this connector. The required block type must be assigned to this tag as shown on Figure 4-4: Setting type for typed connector.

![Figure 4-4: Setting type for typed connector.](image)

In the example, the type of connector is Cable_Pow (Power Cable). The meaning is that the paths between the items in the technical layer will not be just some connectors, but each will be a part of block Cable_Pow, and the attribute values of these connectors will be filled in an appropriate way. The most important is that the plug-in and the optimization will fill the distances that the cables cover, using the geometrical data from the database, making it possible for us to know, for example, the weight and the cost of the cables.

4.3.4 Optimization model parameters
Parts, links, dependencies and attributes that haven't <<optimized>> stereotype are treated as optimization model parameters. Parameters can be taken directly from the model or from the related database. For those parameters which values are taken from database all corresponding SysML elements must have <<inventory>> stereotype. Setting <<inventory>> stereotype on part, link or dependency implies that corresponding element represent some table of corresponding "physical" elements in the database. Each row of this table must include values of <<inventory>> attributes of corresponding element.

Concise model also allow creating auto-calculating parameters. The values of auto-calculating parameters is not known during modelling process but can be calculated from values of other parameters using some mathematical formulas. To create auto-calculated parameter <<optimized>> and <<derived>> stereotypes must be applied to the attribute represented corresponding parameter. The value of this attribute calculated using attached constraint that typically have <<sow_assignment>> stereotype (other concise constraint stereotypes are also can be applied). On the Figure 4-5: Usage of auto-calculating parameters we can see different auto-calculating parameters representing length, cost and weight of power cable. The values of these parameters depending from actual cable length, i.e. calculated from distance of geometrical route to which "physical" cable can be allocated. Formulas for the parameters calculation are shown in the corresponding constraints.

![Figure 4-5: Usage of auto-calculating parameters](image)

### 4.3.5 Decision variables

All parts, links, dependencies and attributes having <<optimized>> stereotype (and not having <<derived>> stereotype) are treated as decision variables. Particular "physical" part, link or dependency can be realized or not realized in the optimal architecture according to the decision of the optimization engine. Concise plug-in add decision variable to each element having stereotype <<optimized>>. This variable gets value 1 if corresponding "physical" elements realized in the optimal architecture and 0 otherwise. The variable is not a part of the concise model, but can be used in the optimization constraints. The variable can be accessed by using <isChosen>, <isSelected> or <part> name.
4.3.6 Optimization goals

An attribute having <<sow_goal_attribute>> became one of the optimization goals. Multiply optimization goals can be defined for the one model. In this case optimization engine instead of one optimal solution calculates set of optimal solution called Pareto frontier. On the Figure 4-6: Usage of multiply optimization goals we have three different optimization goals (optimize system cost, system weight and total cable length).

The attribute marked by this stereotype equipped with following tags:

- **Action**: can be minimize or maximize depending on optimization goal.
- **Description**: string described the optimization goal.
- **isEnabled**: can be true or false. Setting this attribute to be false "turned off" selected optimization goal.
- **Priority**: set priority for selected optimization goal.

Example of settings for these tags is shown on Figure 4-7: Setting tags of optimization goal attribute. In the example the optimization goal is to minimize system cost and this goal have high priority (priority=1).
4.3.7 Explicit constraints

Constraints of the optimization model can be defined in the explicit or implicit form. Explicit constraints are defined using SysML constraint element. The corresponding constraint element must have <<sow_optimization>> stereotype. Additionally <<sow_constraint>> stereotype can be applied to optimization constraint. The attribute marked by <<sow_constraint>> stereotype equipped with following tags:

- Description: string described the constraint.
- isEnabled: can be true or false. Setting this attribute to be false "turned off" selected constraint.
- isVisible: can be true or false.

Example of settings for these tags is shown on Figure 4-8: Setting tags for optimization constraint.

Explicit constraints are defined by using GSCL extension for optimization (see D6.3.2 Specification of the goal contracts specification language (DANSE Consortium, 2013)).
4.3.8 Implicit constraints

Implicit constraints are not specified in the concise model by using some specific SysML element, but translated into optimization code from concise model mappings, multiplicities and from stereotypes related to special algebras.

4.3.8.1 Constraints from multiplicities

Each part in the internal block diagram has multiplicity. The multiplicity can be set as a single number (1), interval of numbers (1..10) or any number (*). In case when multiplicity defined as number or interval of numbers it is automatically translated into constraint in the optimization code. I.e. if some part have multiplicity 1, then there is exactly one element of this type must be in the optimal architecture, or if some part have multiplicity 1..10, then optimal architecture can't have less then one or greater then ten corresponding elements.

Each link also has multiplicities on its ends. These multiplicities are defined in the same way as for the parts. In this case meaning of multiplicities is number of links between corresponding parts. I.e. if on the one end of the link we have multiplicity 1 and on the other end we have multiplicity *, then each element from the second end must be connected to exactly one element from the first end.
4.3.8.2 Constraints from special algebras

Optimization constraints are also created from stereotypes related to special algebras. For example in the communication system scenario (see D8.3.1 Prototype I) number of generic antennas connected to one generic controller dependent from technical characteristics of the particular controller and defined by the <MaxCapacity> attribute. The controller is chosen from the catalogue of generic controllers and value of the corresponding attribute is not known during modeling process, so we can't use multiplicities mechanism to define constraints on the maximal number of connections to one controller. But we can use special interface algebra for this purpose. Setting <<interfaceMaxCapacity_controllerGeneric>> stereotype on the attribute <MaxCapacity> attribute of generic controller and <<interfaceRequest_controllerGeneric>> stereotype on the link between generic antenna and generic controller "turns on" the special interface algebra, and we obtain necessary optimization constraint during model translation performed by the concise plug-in. The part of the communication system model with related elements shown on the Figure 4-9: Usage of interface special algebra

![Figure 4-9: Usage of interface special algebra](image)

The full list of the special algebras and corresponding stereotypes can be found in the D6.5.2 Extension of standard profiles for DANSE Modelling ([DANSE Consotium, 2014]).
5 Conclusion

In this document the important SoS modeling characteristics are described on an conceptual level. A detailed meta model is given illustrate how these concepts can be implemented as an modeling language. For practical application we recommend to use the already existing and used UPDM extended by the means provided with D6.5.2. Together with the reuse of existing models in UPDM, the extensions and the concepts defined in the DANSE modeling formalism, different types of analysis can be performed. The contract concept gives a clear semantics to the behavioral, evolution and even structural part of modeling SoS.

The concepts and their implementations/realizations presented in this document are used in the DANSE project for different use cases, analysis and optimization methods. Furthermore it allows to build upon existing models and is easy to integrate in other modeling languages if e.g. the DANSE semantic mediation methodology is applied.

The combination of analysis of current and future SoS models, reconfiguration of the SoS and specifications for both aspects integrated into one formalism enables the methodology “Correct-by-Evolution” which is the guiding theme of the DANSE project.
# 6 Abbreviations and Definitions

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAE</td>
<td>DANSE Concept Alignment Example</td>
</tr>
<tr>
<td>CS</td>
<td>Constituent System</td>
</tr>
<tr>
<td>DANSE</td>
<td>Designing for Adaptability and evolution in System of Systems Engineering</td>
</tr>
<tr>
<td>DMM</td>
<td>DANSE meta model</td>
</tr>
<tr>
<td>DoDAF</td>
<td>Department of Defence Architecture Framework</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>MODAF</td>
<td>Ministry of Defence Architectural Framework</td>
</tr>
<tr>
<td>SoS(s)</td>
<td>System(s) of Systems</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>UPDM</td>
<td>Unified Profile for MoDAF and DoDAF</td>
</tr>
</tbody>
</table>
7 References


(January 2012). Unified Profile for DoDAF and MODAF. OMG.


8 Appendix I: Meta Model

The set of core modeling artifacts is illustrated in Figure 2-1: Core Concepts. The “ConstituentSystem” has a “satisfy” relation to a set of Contracts which represent the specification of the CS. The specification of each CS is based on the GCSL (refer to D6.3.2 GCSL specification deliverable) based on the contract approach. This is represented by the “assume” and “guarantee” attribute of the CS which are constraint to contain the assumption and guarantee as natural text or after the formalization as GCSL statements. The implementation of a CS refers to the real world system or any kind of model for this. The DANSE simulation framework e.g. expects a FMU which represents the implementation during simulation.

CSs may contain other CSs and this containment relation is expressed in both directions with the relations to the “owner” and to the set of owned CSs.

The “Exchange” element covers communication and the exchange of other “Resources” like “Matter”, “Energy” and “Information” (not illustrated in Figure 2-1: Core Concepts). Each exchange is characterized by one “sender” and at least one “receiver” side of the exchange. We explicitly allow broadcasting to model information exchange.

In Figure 2-2: Advanced Concepts the refined types of ConstituentSystem are presented in light grey. The “Infrastructure” sub-type represents those CSs which are passive in the sense that they do not follow any goal. These CSs represent typically infrastructures or resources which are used by the active CSs (“ActiveConstituentSystem”) in order to improve a goal (“Goal”). Active CSs are on the “user” side of the “Use” relation while on the other only passive CSs are allowed. Semantically the used CS is treated like a resource required to satisfy the contracts or to improve the goals of the user. Beside the “user” and “used” roles implied by the “use” relation other roles help to define what is required from an CS to participate in an SoS. These “Roles” are characterizing the instances of CS an SoS. The CS themselves may participate in different SoSs but in each they assume a certain SoS-specific role. Roles are defined per SoS and restrict the behavior of the participating CS. A role is specified via a set of contracts and this allows checking if a CS is able to fulfill the role by checking if the role specification is contained in the CS specification.

8.1 Common Package

8.1.1 Entity

The abstract “Entity” serves as utility element providing basic attributes for most other element. For all elements it’s technically useful to be able to associate a name, path including the name and some id.

Super-Types: none

Attributes:

- [0,1] name : String
- [0,1] qualifiedName : String

This attribute is used to store the full qualified name / path of the element in the model.
8.1.2 Unit

As “Unit” one of the Si-units (International System if Units) should be used.

**Super-Types:**

- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String

**Attributes:**

- none

**References:** none

8.1.3 Resource

The abstract “Resource” represents all exchange items. The combination of quantity and unit allows defining e.g. flows with capacity X and Y as unit (→ 3 kilogram).

**Super-Types:**

- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String

**Attributes:**

- [0,1] quantity : double

**References:**

- [0,1] unit : Unit

8.1.4 Information

The exchange item “Information”. It abstracts from the physical artifact carrying the data. In contrast to other exchanged item information might be not reduced or consumed by the exchange. If one exchanges his/her name with someone else the information “what is his/her name” is shared, if one exchanges confidential information the attribute “confidential” probably is lost by the exchange.

**Super-Types:**

- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
8.1.5 Event

The exchange item “Event”. An event is a trigger for a specific Activity to be performed or enabled. The inherited attributes “quantity” and “unit” are constraint to be null/empty/not set.

Super-Types:

- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String

- Resource
  - [0,1] quantity : double
  - [0,1] unit : Unit

Attributes: none
References: none

8.1.6 Matter

The exchange item “Matter”.

Super-Types:

- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String

- Resource
  - [0,1] quantity : double
  - [0,1] unit : Unit

Attributes: none
References: none
8.1.7 Energy
The exchange item “Energy”.

Super-Types:
- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String
- Resource
  - [0,1] quantity : double
  - [0,1] unit : Unit

Attributes: none
References: none

8.2 Core Package

8.2.1 ConstituentSystem
The abstract “ConstituentSystem” is the base type of all CSs in the modeling. All these elements have three
thinks in common. They could be arranged in a hierarchical containment relation, have observable
input/output variables, a contract-based specification and an implementation. The containment relation gives
the typical (de-)composition structure like in component based system engineering. The input/output
variables are simplified as the index of the list of input or output resources. Contracts specify the behavior of
the CS and refer to the interface of the CS or to the architectural structures and conditions of the
environment of the CS which includes other CSs as well. The implementation of the CS is the real world
system that actually exists without any model but which could be a model itself. The variability of those
entities is quite high and the specification only requires the black box view of those CS, a link to the
implementation is useful enough in early phases. In later phase not the implementation is relevant, it is only
relevant if the implementation actually satisfies the specification.

Super-Types:
- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String

Attributes:
- [0,1] implementedBy : String
  This is a textual representation of a link to the implementation of the CS. A implementation could be

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Date</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Final</td>
<td>2014-07-31</td>
<td>47</td>
</tr>
</tbody>
</table>
given in different formats (UML diagram, source code …). In DANSE the simulation framework e.g. requires a FMU as such implementation.

References:

- [0,*] participates: SystemOfSystems
  The list of SoSs where the CS is participating in.
- [1,*] specifications : Contract
  The set of contracts the implementation of the CSs shall satisfy.
- [0,*] satisfies : Contract
  The set of contracts the implementation of the CSs satisfies.
- [0,1] owner : ConstituentSystem
  A link to the owner of this CS.
- [0,*] owns : ConstituentSystem
  A containment relation to the parts of the CS.
- [0,*] inputs : Resource
  The set of Resources consumed by the CS.
- [0,*] outputs : Resource
  The set of Resources emitted by the CS.

8.2.2 Contract

The “Contract” is the specification of a CS. This specification is separated into assume-guarantee-pairs which restrict the environment of the CS – which are the prerequisites of participating in the SoS – and the restrictions of the behavior of the CS itself. The combination of both allows verifying the CS independently from the SoS and thus enables reusability of verification results. To tackle the complexity issues in the domain of SoS each contract can be associated with a satisfaction probability threshold. This enables to use e.g. statistical model checking to verify the satisfaction relation. Between different contracts a refinement relation may exist (see (Damm, Hungar, Josko, Peikenkamp, & Stierand, 2011) for details).

Super-Types:

- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String

Attributes:

- [1] probability :Double
  Gives the probability threshold that the contract is fulfilled by the CS.
• [0,1] assume : String
  The assumption as text-pattern in GCSL language.

• [1] guarantee : String
  The guarantee as text-pattern in GCSL language.

References:
• [0,*] refinedBy : Contract

8.2.3 Goal

A “Goal” represents the optimization target of an active CS or SoS.

Super-Types:
• Entity
  o [0,1] name : String
  o [0,1] qualifiedName : String
  o [0,1] id : String

Attributes:
• [1] isMinimize : Boolean
  Gives the optimization “direction”.

• [1] isEnabled : Boolean
  Goals might be not active e.g. depending on environmental conditions.

• [1] priority : Integer
  Each goal has a certain priority. Several goals might have the same priority.

References:
• [0,*] refinedBy : Goal
  The set of decomposed goals which all together imply the goal.

8.2.4 Infrastructure

The passive ConstituentSystem is called “Infrastructure”.

Super-Types:
• Entity
  o [0,1] name : String
  o [0,1] qualifiedName : String
  o [0,1] id : String

• ConstituentSystem
  o [0,*] participates : SystemOfSystems
8.2.5 ActiveConstituentSystem

The active CS follows its own goals and interacts actively with other CSs in terms of coordination.

Super-Types:

- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String
- ConstituentSystem
  - [0,*] participates: SystemOfSystems
  - [1,*] specifications : Contract
  - [0,*] satisfies : Contract
  - [0,1] owner : ConstituentSystem
  - [0,*] owns : ConstituentSystem
  - [0,*] inputs : Resource
  - [0,*] outputs : Resource

Attributes: none

References:

- [1,*] objectifies : Goal
  The set of goals the active CS targets to.
- [0,*] commands : ConstituentSystem
  The set of CS that are commanded by the CS.
- [0,*] hasAuthorityOverCS: ConstituentSystem
8.2.6 SystemOfSystems

The “SystemOfSystems” represents a CS which is itself a SoS. This can be used to define the current project but also to refer to other SoSs as participating CSs. A SoS contains a set of Roles and Rules for the participating CSs which follow/assume such Rules and Roles.

Super-Types:

- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String

- ActiveConstituentSystem
  - [0,*] participates: SystemOfSystems
  - [1,*] specifications : Contract
  - [0,*] satisfies : Contract
  - [0,1] owner : ConstituentSystem
  - [0,*] owns : ConstituentSystem
  - [0,*] inputs : Resource
  - [0,*] outputs : Resource
  - [1,*] objectifies : Goal
  - [0,*] commands : ConstituentSystem
  - [0,*] hasAuthorityOverCS: ConstituentSystem

Attributes:

- [0,1] name : String

References:

- [0,*] relations : Exchange
- [0,*] rules: Rule
- [0,*] roles: Role
- [0,*] participating: ConstituentSystem

8.2.7 Exchange

The “Exchange” of information, event, matter or energy between at least two CSs is modeled as a 1:n relation. The sender is unique but the receivers could be multiple CSs.

Super-Types:
8.2.8 DynamicityContract

The “DynamicityContract” allows to restrict the evolution of the SoS in terms of reconfiguration of the SoS (see section 2.3 for details).

**Super-Types:**

- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String

- Contract
  - [1] probability : Double
  - [0,1] assume : String
  - [1] guarantee : String
  - [0,*] refinedBy : Contract

**Attributes:**

- [0,1] condition : String
  GCSL property which guards the transition from the assumed (left hand) to the guaranteed (right hand) of evolution steps.

**References:** none
8.3 Advanced Concept Package

8.3.1 Activity

An activity transforms a set of resource into another one while potentially changing the physical environment. The resources are exchanged between the CS but there is also an interaction with the physical world. One example is to carry a load (resource) from one CS to another. The other one might be at a different location and to carry changes the environmental conditions. Activities can be defined without the allocation to an CS but at some point in the refinement process the required capabilities (“to be able to carry an load”) have to be allocated to the activity. Later on the activity can be allocated to every CS that has the right interface and capabilities. Contracts are used to specify the relation between input and output of an activity.

Super-Types:
- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String

Attributes:
- [0,1] name : String

References:
- [0,1] performedBy : ConstituentSystem
- [0,*] specification: Contract
- [0,*] inputs: Resource
- [0,*] outputs: Resource
- [0,*] requiredCapabilities: Capability

8.3.2 Service

Services are complex activities. These combine several “steps” in a potentially branched network. Depending on the condition at a certain point in time one of the next steps is taken. Services might consist of activities performed by different CSs and therefore it potentially depends on the cooperation of those CS. The network of activities is named “plan” because it contains assumptions on future states and activities.

Super-Types:
- Entity
  - [0,1] name : String
DANSE Modelling Formalism, including Domain Metamodel & Semantics: Focused on support for analysis and optimization

- [0,1] qualifiedName : String
- [0,1] id : String

- **Activity**
  - [0,1] performedBy : ConstituentSystem
  - [0,*] specification: Contract
  - [0,*] inputs: Resource
  - [0,*] outputs: Resource
  - [0,*] requiredCapabilities: Capability

**Attributes:**
- [0,1] name : String

**References:**
- [1] plan : Network

### 8.3.3 Network

The network is the structure for combining activities to a plan. It consists of at least one transition, the default transition.

**Super-Types:**
- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String

**Attributes:**
- [0,1] name : String

**References:**
- [1] defaultTransition : Transition
- [1,*] transitions : Transition

### 8.3.4 Transition

Transitions combine conditions (guards) with actions (activities). Each transition has a potentially empty set of follow up transitions. Semantically a transition can only be taken if the guard is evaluated to true. No next transition means the network reach a final state.

**Super-Types:**
- Entity
  - [0,1] name : String

---

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Date</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Final</td>
<td>2014-07-31</td>
<td>54 of 58</td>
</tr>
</tbody>
</table>
o  [0,1] qualifiedName : String
o  [0,1] id : String

Attributes:

•  [0,1] guard : String

References:

•  [0,*] actions : Activity
•  [0,*] nextTransitions : Transition

8.3.5 Rule

A rule is a constraint on the behavior of CSs in a SoS. It is defined for roles instead of certain CSs because the rules do not depend on the CS but the CS have to obey them under certain conditions. A specification can be given as natural language and refined using contracts.

Super-Types:

•  Entity
  o  [0,1] name : String
  o  [0,1] qualifiedName : String
  o  [0,1] id : String

Attributes:

•  [1] specification : String

References:

•  [0,*] refersTo : Role
•  [0,*] refinedBy : Contract

8.3.6 Role

A role represents an abstract CS participating a SoS. The behavior is foreseen as a network of activities (like a service) refined by a set of contract or a set of rules. For most roles a set of goals is foreseen to be followed by the active CSs assuming that role.

Super-Types:

•  Entity
  o  [0,1] name : String
  o  [0,1] qualifiedName : String
  o  [0,1] id : String

Attributes: none

References:
8.3.7 Capability

The ability to perform an activity is called capability. Those are exposed by a CS as an abstract view of its behavior.

Super-Types:

- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String

Attributes:

- [0,1] name : String

References:

- [0,*] exposedBy : ConstituentSystem

8.3.8 Strategy

An active CS may follow a plan of activities in order to fulfill a set of goals. This plan is called strategy. Strategies are independent from CS, i.e., they exist even if they are (currently) not performed by an CS.

Super-Types:

- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String

Attributes:

- [0,1] name : String

References:

- [0,*] addressedGoals : Goal
- [1] plan : Network
- [0,1] performedBy : ConstituentSystem
8.3.9 Use
The use relation is a quite abstract relation between an active and a passive CS, where the active one uses the passive like a resource.

Super-Types:
- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String

Attributes:
- [0,1] name : String

References:
- [1] user : ActiveConstituentSystem
- [1] used : ConstituentSystem
- [0,1] userRole : Role
- [0,1] usedRole : Role

8.3.10 Authority
The authority is a special kind of CS that is able to define rules and thereby roles of the SoS. Typically this CS are authorities that define regulations or laws.

Super-Types:
- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String

Attributes:
- [0,1] name : String

References:
- [0,*] hasAuthorityOver : Rule

8.3.11 Knowledge
Knowledge is what is extracted from information and observation of the environment. In this domain only the quite abstract concept is realized because a detailed model of cognitive processes is beyond the scope of the DANSE project.

Super-Types:
8.3.12 WorldModel

Each individual CS has its own view to the world. This view is created from past exchange with other CS and the environment. All knowledge is an interpretation of the observed information or other exchanged resources. The geographical distribution limits this view to the other participating CS but communication infrastructures compensate parts of this awareness.

Super-Types:

- Entity
  - [0,1] name : String
  - [0,1] qualifiedName : String
  - [0,1] id : String

Attributes:
- [0,1] name : String

References:
- [0,*] facts : Knowledge
- [0,*] localViewOf: ConstituentSystem