### Deliverable Information

<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>287716</td>
</tr>
<tr>
<td>Deliverable ID</td>
<td>D_6.1</td>
<td>Date</td>
<td>2012-05-04</td>
</tr>
<tr>
<td>Status</td>
<td>Final</td>
<td>Version</td>
<td>Final</td>
</tr>
<tr>
<td>Contact Person</td>
<td>Axel Legay</td>
<td>Organisation</td>
<td>INRIA</td>
</tr>
<tr>
<td>Phone</td>
<td>E-Mail</td>
<td><a href="mailto:Axel.Legay@inria.fr">Axel.Legay@inria.fr</a></td>
<td></td>
</tr>
</tbody>
</table>
INTRODUCTION ................................................................................................................................. 5

1.1 Existing Classifications .................................................................................................................. 5
  1.1.1 SoS Definitions ............................................................................................................................. 5
  1.1.2 Maier Characteristics .................................................................................................................. 6
  1.1.3 SoS Classification ........................................................................................................................ 6

1.2 Approaches .................................................................................................................................... 7
  1.2.1 Current State-of-the-Art Approaches ........................................................................................... 7
    1.2.1.1 Architecture Solutions ........................................................................................................... 7
    1.2.1.2 Integration Solutions ............................................................................................................ 8
    1.2.1.3 Collaboration Solutions ........................................................................................................ 9
    1.2.1.4 Test and Evaluation Solutions .............................................................................................. 9
  1.2.2 Graph Transformation Systems ................................................................................................ 10
  1.2.3 Dynamic Communicating Systems ............................................................................................ 13
  1.2.4 UPDM (MoDAF, DoDAF) .......................................................................................................... 14
    1.2.4.1 Introduction ............................................................................................................................ 14
    1.2.4.2 The history of DoDAF, MODAF and NAF ............................................................................ 15
    1.2.4.3 The Unified Profile for DoDAF and MODAF (UPDM) ............................................................ 16
    1.2.4.4 The rationale behind UPDM .................................................................................................. 16
    1.2.4.5 UPDM 2.0 Compliance Levels .............................................................................................. 17
    1.2.4.6 DOD/MOD support for UPDM and international standardisation ......................................... 18
  1.2.5 Multi Agents .............................................................................................................................. 18
    1.2.5.1 The main concepts definition ............................................................................................... 18
    1.2.5.2 The methodology steps ........................................................................................................ 19
    1.2.5.3 Synthesis and added value for SoS (vs.UPDM) ..................................................................... 20
    1.2.5.4 KAOS : A goal oriented methodology .................................................................................. 21
      1.2.5.4.1 Main concepts ..................................................................................................................... 21
      1.2.5.4.2 Main views and models ................................................................................................... 22
      1.2.5.4.3 Typical workflow ............................................................................................................. 22
      1.2.5.4.4 Synthesis and added value for SoS (vs.UPDM) ................................................................. 23
  1.3 Summary ......................................................................................................................................... 24

1.4 Specific requirements ...................................................................................................................... 25
  1.4.1 RQ-05-004 .................................................................................................................................. 25
  1.4.2 RQ-05-011 .................................................................................................................................. 25
  1.4.3 RQ-10-007 .................................................................................................................................. 25
  1.4.4 RQ-05-024 & RQ-05-025 ............................................................................................................. 25
  1.4.5 RQ-05-006 & RQ-05-029 ............................................................................................................. 26
  1.4.6 RQ-10-002 .................................................................................................................................. 26
  1.4.7 RQ-10-003 .................................................................................................................................. 26
  1.4.8 RQ-05-001 .................................................................................................................................. 26
  1.4.9 RQ-10-009 .................................................................................................................................. 26

2 PROPOSAL FOR A COMPREHENSIVE DANSE APPROACH FOR SOS ......................................................... 27
  2.1 Test and Evaluation Solutions ....................................................................................................... 27
  2.2 Graph-Transformation Systems ................................................................................................... 30
    2.2.1 Some considerations regarding UPDM ....................................................................................... 30
      2.2.1.1 Capabilities ............................................................................................................................. 30
      2.2.1.2 Services .................................................................................................................................. 32
      2.2.1.3 Operational views ................................................................................................................ 34
      2.2.1.4 System views ........................................................................................................................ 35
      2.2.1.5 Discussion ............................................................................................................................. 37
      2.2.1.6 UPDM Gaps vs. specific requirements .................................................................................. 37
    2.2.2 Graph Grammars (GG), a framework for dynamic SoS architectures ..................................... 38
      2.2.2.1 Principles and overall objectives ......................................................................................... 38
      2.2.2.2 Sketching an illustrative example: the “rescue” SoS ............................................................ 38
      2.2.2.3 GG rules to establish the “constitution of rescue SoS” ........................................................ 40

Version Status Date Page
0.00 Draft 2012-06-04 2 of 57
2.2.2.4 Adding guards to control run time architecture evolutions .................................................. 43
2.2.2.5 How to implement GG and GG rules? ................................................................................. 45
2.2.3 Some questions regarding concepts for reuse ......................................................................... 46
  2.2.3.1 Services in SoS: anything specific? ...................................................................................... 46
  2.2.3.2 Systems in SoS: anything specific? ...................................................................................... 47
  2.2.3.3 Contracts in SoS: how do they survive? .............................................................................. 47
2.2.4 Goals: a proposal .................................................................................................................... 47
  2.2.4.1 What goals should offer as a concept .................................................................................. 48
  2.2.4.2 Proposed approach: goals as Gibbs fields on graphs .......................................................... 48
  2.2.4.3 Distributed and Autonomic goal optimization  .................................................................. 49
  2.2.4.4 Comments on this approach to handle goals ................................................................. 49
2.2.5 Contribution and positioning with respect to the five dimensions and other remarks .......... 50
  2.2.5.1 Operational independence vs. Managerial independence .................................................. 51
  2.2.5.2 Evolutionary development .............................................................................................. 51
  2.2.5.3 Emergent behaviour ..................................................................................................... 51
  2.2.5.4 Geographic distribution .................................................................................................. 52
  2.2.5.5 Other remarks .................................................................................................................. 52
2.3 Dynamic Communicating Systems ............................................................................................ 52
2.4 Multi-agents ............................................................................................................................... 53

3 BRIEF CONCLUSION .................................................................................................................... 55

4 REFERENCES .................................................................................................................................. LVI
Content of Figures

Figure 1. Interpretive Measurement of "SoS-ness" ................................................................. 6
Figure 2 .................................................................................................................................... 11
Figure 3 ....................................................................................................................................... 11
Figure 4 ....................................................................................................................................... 12
Figure 5 ....................................................................................................................................... 12
Figure 6 Merge manoeuvre and disappearance of one free agent ........................................... 13
Figure 7 Partner Abstraction ....................................................................................................... 14
Figure 8 Spotlight Abstraction .................................................................................................... 14
Figure 9 UPDM Compliance levels L0 and L1 ......................................................................... 17
Figure 10 ..................................................................................................................................... 31
Figure 11 ...................................................................................................................................... 32
Figure 12 ...................................................................................................................................... 33
Figure 13 ...................................................................................................................................... 34
Figure 14 ...................................................................................................................................... 34
Figure 15 ....................................................................................................................................... 35
Figure 16 ....................................................................................................................................... 35
Figure 17 ....................................................................................................................................... 36
Figure 18 ....................................................................................................................................... 36
Figure 19 ....................................................................................................................................... 41
Figure 20 ....................................................................................................................................... 42
Figure 21 ....................................................................................................................................... 42
Figure 22 ....................................................................................................................................... 44
Figure 23 ....................................................................................................................................... 48
1 Introduction

In this deliverable the gap between the existing modelling languages/methodologies and the industrial needs is identified. The result of this analysis is one building block for the further progress of the development of the DANSE modelling methodology and the associated language and analysis capabilities. It has direct impact to the modelling formalism, the goal and contract specification language and the extension of existing modelling profiles like UPDM.

The industrial needs are derived from the DANSE requirements which had been identified in parallel as the state of the art analysis. The state of the art analysis is based on the background of the academic and industrial project partners and complemented by literature research.

The following section of deliverable starts with a review of classifications for System of Systems (SoSs). The second part states the different modelling approaches. UPDM is a profile to implement the MoDAF and DoDAF data models and their methodology and since this is not part of the state of practice this is also mentioned as “UPDM approach” to simplify the wording. In section 3 a subset of the DANSE requirements is presented. These requirements are directly or indirectly related to the modelling formalism and represent those needs which are in the scope of DANSE. The fourth section contains the gap analysis itself where the different approaches are applied to the DANSE SoS characteristics (section 2) and the synthesis of this analysis. The last section is the conclusion.

1.1 Existing Classifications

1.1.1 SoS Definitions

Systems of systems were first recognized as distinct entities in the 1990s, as systems and their interactions became increasingly complex. With this increase in complexity, it became apparent to some far-thinking individuals that there was important behaviour, of interest to stakeholders, that occurred at a level above the constituent systems. With such behaviour, it became worthwhile to consider the aggregation of systems into something larger that embodied that behaviour. A first idea in this regard was inherent in [Manthorpe 1996], as a definition:

A system of systems (SoS) is a system comprised of elements that are systems.

This understanding has become widespread, but is only partially useful. While it expresses the basic idea, the interpretive definition of the word system can apply this definition to nearly any system. Even a microprocessor is comprised of lower-level “systems.”

Other authors have tried to make the definition more precise by adding additional thoughts:

Systems of systems are large-scale concurrent and distributed systems that are comprised of complex systems. [Jamshidi 2005]

Jamshidi adds the concepts of “large-scale” and “concurrent and distributed” as applied to the SoS, while also adding the concept of “complex” to the constituent systems. These additional concepts would exclude a microprocessor, but still leave much room for interpretation.

A system of systems is defined as a set or arrangement of systems that results when independent and useful systems are integrated into a larger system that delivers unique capabilities. [US DoD 2006]

The writers of the US DoD SoSE Guide add the concept of the emergent behaviour (“unique capabilities”) of the SoS itself, as well as characterizing the constituent systems as “independent” and “useful.” Again, this would exclude the microprocessor because its constituent systems have little use outside the microprocessor – but the microprocessor certainly delivers unique capabilities.

The problem with nearly all such SoS definitions is that they have trouble distinguishing between a system and a system of systems. The concept of system is itself so general as to create this difficulty. Without a clear distinction, the entire concept of system of systems suffers from muddy thinking at its outset.
1.1.2 Maier Characteristics

A solution to the distinction between system and system of systems was offered by [Maier 1998] in a recognition that it is a series of characteristics that make the difference. In addition, these characteristics are not binary. A system becomes more and more like a system of systems as the characteristics become more evident. Maier’s characteristics are given as

- **Emergent behaviour** – A SoS performs functions that are not achievable by any of the independent component systems. Usually, these functions are only evident at a higher level of perception and do not even make sense at the level of the independent component systems. This particular characteristic is true for both systems and SoS; its presence in the list ensures that the SoS in fact has an existence above and beyond the constituent systems.

- **Geographic distribution** – The constituent systems are distributed in such a way that geographic issues affect the interactions between them – both operational interactions and managerial interactions.

- **Evolutionary development** – Functions and purposes of the SoS are added, removed and modified in an ongoing way. Often, the SoS functions change due to changes in the constituent systems. Such changes, happening asynchronously, result in constant evolution of the SoS.

- **Operational independence** – The constituent systems each have their own independent purposes. If detached from the SoS, the constituent systems are still useful.

- **Managerial independence** – The constituent systems are developed and managed for their own purposes. Relatively unrelated owners with little coordination may perform the management.

These characteristics exist to a greater or lesser extent in each system. The microprocessor example used earlier has emergent behaviour, but has little of the other four characteristics. As systems become larger and more complex, with constituent elements operating as systems in their own rights, the “SoS-ness” of the overall system becomes more apparent. Figure 1 shows subjective measurements of four SoS examples, in which an airport has low SoS characteristics while the Internet is at the top of all five characteristics. The light-colored area represents the SoS characteristics of interest to DANSE.

1.1.3 SoS Classification

Another way to classify the SoS is also provided by [Maier 1998]. While still recognizing the continuum of measurement in the five characteristics, Maier also found it useful to group SoS into these three major types:

- **Directed SoS** – The SoS is built and managed to fulfil specific purposes, typically by a controlling authority or primary stakeholder. During its long term operation, the SoS is centrally managed by that controlling authority. The constituent systems are independent, but have a clear subordinate relationship to the SoS.
A typical example is the Air Traffic Control SoS within any one country. The controlling agency plans, acquires, and manages all systems to provide ATC functions. Each system provides specific functions such as radar imaging or communications, and the entire set of systems provides safety and general air traffic control.

- **Collaborative SoS** – The SoS develops over time through collaborative work by the stakeholders of the constituent systems. As the SoS value becomes evident, the stakeholders may identify and implement a central management agency, but often that agency has no coercive power. Instead, the agency performs its management function by influence toward the stakeholders. The stakeholders and constituent systems voluntarily collaborate for the higher-level benefits received through the existence and improvement of the SoS. A typical example is the airport, in which many stakeholders have an interest. The constituent systems at the airport may be owned by different agencies and are planned, acquired, and managed for the purposes of those agencies. Nonetheless, collaborative interfaces among the systems are implemented to enhance the overall airport functions of smooth passenger flow and security.

- **Virtual SoS** – In this type of SoS, there is no central manager and no central authority. Individual systems interact to perform their intended purposes, and the inherent value of the SoS becomes apparent to each stakeholder. There is no “agreed” SoS purpose, but stakeholders realize the benefit of the higher-level purposes that develop. Each stakeholder modifies the constituent systems to enhance their perception of the SoS value. As a result, large-scale behavior emerges and is desirable to the various stakeholders. A typical example is supply chain management, in which individual companies interact with each other to create a complex network of “preferred suppliers.” The companies acquire and enhance software and systems to use the network. Each such installation adds to the overall SoS capability for all.

### 1.2 Approaches

#### 1.2.1 Current State-of-the-Art Approaches

The usual approaches to SoS management today are based heavily in current thoughts of Program Management (PM) and Systems Engineering (SE), treating the SoS as a “large system.” These approaches generally work, but have often generated untoward surprises. As a result, new paradigms are developing to enhance the usual PM and SE approaches for these “very large systems.”

#### 1.2.1.1 Architecture Solutions

Architecture solutions are characterized by a higher-level view of the SoS in relationship to its operation, functionality, and constituent systems. They often entail the use of models, either conceptual or executable, that represent the SoS to allow human understanding of the complexity.

**Capability engineering** is one architectural approach, based heavily in traditional top-down systems engineering concepts. In this approach, the SoS is envisioned to perform certain user-desirable services and functions, known as “capabilities.” The SoS capabilities can be mapped and analyzed for compatibility and completeness using various relational diagrams, then allocated downward into the requirements of the constituent systems. During the allocation, additional diagrams provide methods to check and coordination the allocation to affirm the allocation effectiveness. Capability engineering is used primarily in military acquisition and is guided by a set of evolving “capabilities documents” that encompass the current version of capabilities. Capability engineering is hampered, however, by the complexity of the SoS itself; the mapping from high-level capabilities to detailed design aspects of the constituent systems results in a modeling complexity that has so far outstripped the available tools.

**Evolutionary re-architecting** is an adaptive variant on capability engineering that recognizes the ongoing and evolutionary nature of a SoS. In this variant, the capabilities are observed and modified over evolutionary time, using a continuing cycle of observing, assessing, modelling/architecting, and building/modifying.

**Dynamic optimization** is a bottom-up approach to architecting SoS that relies on the motivations of the constituent system stakeholders to improve the SoS for their individual purposes. As stakeholders realize
benefits from the SoS emergent behaviours, they individually and collectively act to improve the SoS. Usually, improvements take the form of modifications to the constituent systems. Sometimes, the stakeholders gather in collaborative forums to coordinate those modifications, or perhaps to coordinate interface capabilities. The result is a constantly-evolving SoS with no entity in charge.

**Architecture frameworks** provide various methodologies and tool sets to support architecture solutions, particularly to address the complexity of a SoS. Architecture frameworks are key to performing either capability engineering or evolutionary re-architecting, and may also be used in collaborative work toward dynamic optimization. An architecture framework is structured to include:

- Standard thought structure
- Standard views and view descriptions
- Standard data structure to retain and relate information
- Standard approach to develop architectures.

By agreeing to such standardization, different teams are offered ways to communicate and collaborate on complex ideas. Some well-known architecture frameworks include Zachman, The Open Group Architecture Framework (TOGAF), the US Federal Enterprise Architecture Framework (FEAF), the US DoD Architecture Framework (DoDAF), and further expansions on DoDAF such as the UK Ministry of Defence Architecture Framework (MoDAF), the NATO Architecture Framework (NAF), and the Unified Profile for DoDAF and MoDAF (UPDM).

**Patterns** provide a means to observe, record, and communicate useful architectural structures. Relying on the innate human ability to recognize and associate patterns, designers identify and document recurring structures that provide specific benefits. When these structures are re-created in other architectures, the benefits should also entail.

### 1.2.1.2 Integration Solutions

Integration solutions are characterized by various control measures on the constituent systems and their interoperability. By controlling the interactions and integration of the constituent systems, the complexity of the SoS can often be managed to some extent.

**Interface control** is a primary means to ensure that constituent systems can interoperate as part of a SoS. Standardized interfaces with strict definitions and rigid certification allow disparate systems to reliably transfer information. This reliable transfer becomes a core basis for SoS operation, preventing failure by ensuring communications among the constituent systems.

**Coupling and interoperability** considerations can often be a strong part of current SoS development. Tight coupling creates systems that depend heavily on each other, while loose coupling allows systems to operate more dynamically under change; usually, loose coupling is preferred in the SoS because the SoS can better adapt to dynamic changes in its environment. Interoperability is a step above interface control, in which designers evaluate the information transferred and its effect on both the SoS and the constituent systems, taking into account the use of that information by recipients.

**Open systems** methodology is a specific form of loose coupling, in which systems are designed to operate within a loose technical framework shared across multiple vendors. The use of open systems allows easier change of the constituent systems within the SoS evolutionary development.

**Commercial Off The Shelf (COTS) integration** is a common approach to open systems, by relying on the commercial marketplace to drive interfaces and compatibility of common elements. COTS integration promises a reduced development cost by amortizing that cost across many commercial applications of an item. Unfortunately, it also brings a lack of control, as the item vendors attempt to meet many conflicting demands with a single item. COTS integration often leads to uncontrolled SoS evolution.

**Legacy system integration** is another common approach to SoS development, in which the SoS is configured using pre-existing “legacy” systems. The SoS often develops itself over time, as the emergent behaviours become apparent to stakeholders, usually without much central management. When central management is introduced to influence the future emergent behaviours, legacy systems are therefore already in place. Further integration (and modification) of these systems is usually much more attractive than new development.
1.2.1.3 Collaboration Solutions

Collaboration solutions are characterized by the “soft systems methodology” of integrating the work of people. With SoS stakeholders and many constituent systems stakeholders, it quickly becomes apparent that SoS management is closely related to people management.

Working with multiple teams is essential in SoS development because of the managerial independence of the constituent systems. Each system has a team associated with it, possibly multiple teams (operational, development, logistic support). With multiple systems in the SoS, this means many teams in dispersed geographic locations, often working in different national cultures. Program management methods apply to specific developmental projects, but ongoing operation/support of the systems may require more adaptive methods. In any case, the coordination and collaboration across the teams presents significant challenge. Even more so than with single projects, project leaders must balance and juggle competing priorities across the teams.

Concurrent systems engineering is a recognition that the development of each constituent system operates at a different pace and in different cycles. The SE cycles used overlap and may even be extended over decades of time. Yet the SoS exists above and beyond the individual systems, with its capabilities constantly evolving as a result of the changes to the constituent systems. SE at the SoS level provides long-term vision for each system, while SE at the system level provides changes that impact the SoS. Maintaining technical integrity in this environment is a significant challenge.

Program interfaces often become key to the coordination of multiple teams. Traditional systems rely heavily on centralized command and control, while the SoS relies much more on influence and indirect control. As a result, it is a common solution to create and maintain specific program interfaces to allow the appropriate influence and control. These program interfaces may be in the form of working groups, Integrated Product Teams, reporting structures, or informal gatherings, as necessary for the specific SoS.

Collaboration tools have been developed to aid in the soft systems methodology. Simple software tools such as meeting scheduling, email communications, and document archives provide significant help. More sophisticated software tools such as web-based conferencing, meeting management, and decision support centers assist multiple teams to communicate and to collaborate. Non-software methods such as positive politics, seeking the win-win scenario, can also be considered tools for collaboration.

1.2.1.4 Test and Evaluation Solutions

Test and evaluation (T&E) solutions are characterized by back-end control of the SoS complexity through the proof of various functions. Test and evaluation occurs at both constituent system and SoS levels; coordination of these various levels provides an opportunity to identify and control emergent behaviour.

Multiple levels of T&E exist within the SoS, and are frequently used in conjunction to determine the SoS effectiveness. Some of the levels that nearly always exist are:

- Interface certification testing
- Constituent system developmental testing
- Constituent system operational testing
- SoS simulations
- SoS testing

Each of these provides a different viewpoint on different hierarchical issues. No single form of T&E provide the full answer; SoS testing does not reveal details that may later deter SoS operation, while lower levels of testing do not exhibit the full SoS capability.

Evaluating interfaces is typically a lowest-level T&E practice that is coupled with interface control as an integration solution. The interface control creates strict definitions for the transfer of information; interface evaluation certifies each system to those definitions. Interface evaluation is often used as a qualifier before allowing a system to “plug in” to the SoS.

Validating functions is essential to SoS verification. Functions exist at both system and SoS levels. T&E occurs on each system change, checking the system against its requirements (which may have been
allocated downward from desired SoS capabilities). As the systems change, the SoS is growing and changing. T&E often also verifies the SoS capabilities during each system change.

**Evaluating dynamics** is often important to the SoS. Even if each system has been proven to its requirements, the dynamic operation of multiple systems through the SoS complexity generates new, unexpected behaviours. The most common method to evaluate dynamics is by observing the SoS and system responses in relation to real events and stimuli. (Simulated events may also be possible in some SoS.) The results reveal variability in system stimulus/response and in SoS responses. Interface noise and different forms of order may be revealed, requiring additional SoS evaluation.

**Evaluating emergence** is an extension to evaluating dynamics. Emergent behaviour is the SoS reason-for-being, to provide some behaviours that are not possible with the individual systems. During ongoing operations, it is useful to plan specific tests and operations to demonstrate the planned emergent behaviour. It is also useful to monitor ongoing operations to discover unplanned emergent behaviours. Such unplanned behaviours may be positive, neutral, or negative. If positive, further SoS changes may be implemented to enhance them; if negative, to mitigate them. One commonly-desired practice is to use executable models with high fidelity to discover and handle unplanned emergent behaviours without suffering their effects on the real SoS. Unfortunately, to date the tools are largely inadequate to provide the needed fidelity at a reasonable cost.

### 1.2.2 Graph Transformation Systems

The following reasons motivate enhancing UPDM with Graph Transformation capabilities:

- The following SoS tasks require dynamic changes in SoS architectures:
  1. Initial Configuration of SoS;
  2. Deployment of new entity (goal, service, system, infrastructure);
  3. Deployment of new instances of existing entities;
  4. Re-configuration to handle alarms, volunteered by the command-and-control center;
  5. Add or modify goals.

- The only means offered by UPDM to capture dynamic changes in architectures are Gantt charts, with change points defined in terms of dates, not events.

**Graph transformation**, or **Graph rewriting**, concerns the technique of creating a new graph out of an original graph using some automatic transformation. It already has numerous applications, ranging from software engineering and software verification to layout algorithms. It is the basis of model transformation techniques extensively used in Model Driven Engineering. We propose to consider this concept for SoS, to capture dynamic changes in architectures. The references below collect the essentials of the theory together with some aspects of its applications.

Following [König2004], graph rewriting is well-suited for the specification of dynamically evolving structures, possessing features such as dynamic creation of objects, mobility and variable topology. Furthermore, graph transformation systems are a very intuitive, natural and general framework. This makes them suitable for an underlying specification language, on which fundamental methods for the verification of dynamically evolving structures can be based. Sometimes **graph grammar** is used as a synonym for graph rewriting system, especially in the context of formal languages; the different wording is used to emphasize the goal of enumerating all graphs from some starting graph, i.e. describing a graph language - instead of transforming a given state (host graph) into a new state.

Transformation rules consist of an original graph, which is to be matched to a subgraph in the complete state, and a replacing graph, which will replace the matched subgraph.
To introduce the matter informally, we simply reproduce the following example of modeling using Graph Transformation systems (GTS), borrowed from the introduction of [König2004]. Consider a system where processes compete for resources $R_1$ and $R_2$. A process needs both resources in order to perform some task. The system is represented as a GTS $Sys$ as follows: graph nodes are standing for ports or connection points used for attaching components. These components are modeled by hyperedges, where a hyperedge has tentacles to all ports it is connected with. In this specific example it is sufficient to consider binary edges only. We consider edges labelled by $R_1;R_2;R_1$ standing for assigned and free resources, respectively, and $P_1$, $P_2$ and $P_3$ denoting the states of a process waiting for resource $R_1$, a process waiting for resource $R_2$ and a process holding both resources, respectively. Furthermore, edges labelled by $D_1$ and $D_2$ represent the demand of a process and connect the target node of a process and the source node of a resource when the process is asking for the resource. When the target node of a resource coincides with the source node of a process, this means that the resource is assigned to the process. The initial scenario for $Sys$ is represented in Figure 2, with a single process $P_1$ asking for both resources.

Figure 2

Figure 3, Figure 4, AND Figure 5 show the three rules of the graph transformation system.

Figure 3

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Date</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>Draft</td>
<td>2012-06-04</td>
<td>11 of 57</td>
</tr>
</tbody>
</table>
For these three figures, reading is as follows. Three graphs are depicted:

- The L (Left) part of the rule is depicted on top of each figure.
- The R (Right) part of the rule is depicted on the bottom of each figure.
- The M (Mid) part of the rule.

Pattern matching for L is performed on the current graph. Pattern matching consists of 1/ matching the graph structure, and 2/ matching the labels on the edges. Each matching results in an auxiliary labeling of the nodes (by 1,2,3,4 in our case) and these nodes are preserved while performing the transformation L → R. As a result, some edges are removed and new edges and nodes are added. This technique of having some invariant graph (it may be more than just nodes) in the middle is called double pushout [Corradini1997] by referring to categorical arguments.
Rule \([\text{AcquireResource}_{ij}]\) shown in Figure 3 can be instantiated with \(i = j = 1\) and \(i = j = 2\) and shows how a free resource \(R_f\) is assigned to a process which subsequently changes its state either from \(P_1\) to \(P_2\) or from \(P_2\) to \(P_3\). A process being in possession of both resources can free them and switch to state \(P_1\) (see rule \([\text{ReleaseResources}]\) on Figure 4). The system described by these two rules has only finitely many states (up to graph isomorphism) and is thus not very challenging. In order to obtain a truly infinite-state system we add rule \([\text{CreateNewProcess}]\) shown on Figure 5 describing the forking of a process creating a new process having access to the same resources.

1.2.3 Dynamic Communicating Systems

Dynamic communication systems (DCS) are a one class of infinite state system. These systems are built up of objects which have an internal state and communicate via messages with other objects. In ([BW, 2006], [BSTW, 2006], [BTW, 2007]) the focus is on the communication topology.

The dynamics of these systems includes the creation and deletion of objects during runtime. Therefore the number of objects is unbounded. The second dynamic aspect is that the communication topologies evolve over time not only due to the fact that new object are created or existing once are removed but also reconfiguration of topology itself. The communication is based on messages and the message queues are in general also not bounded to a specific size. The number of communication protocols is assumed to be bounded and know a-priori as well as the types of messages which could occur. Furthermore, the internal states of the objects only reflect the state with respect to the communication protocol.

The running example in the work mentioned before is “car platooning” inspired by the PATH project [PATH, 2003] and [HESV, 1991]. A car platoon is a dynamic composition of several cars to one unit to maximize traffic density. The car on the head of the platoon is the “leader” and all other cars (of the platoon) are “followers”. Cars not connected to the platoon are “free agents”.

As mentioned before the change of the state (free agent, leader and follower) of the cars depends on the communication and furthermore the appearance (creation) and disappearance (deletion) of objects is part of this example. In Figure 6 a scenario of merging a free agent an existing platoon while another free agent disappears.

One challenging question in this context is for example if a situation is reachable where two cars at the same point in time take the role of the leader for the same platoon. This situation could lead to an accident and therefore the communication protocols have to prevent this configuration.

Due to the unbounded nature of configuration traces this can only be verified using an appropriate abstraction. In [BW, 2006] the representation of the dynamically changing communication topologies by topology graphs and graph transformation rules is described. Also alternative specification approaches like the \(\pi\)-calculus [SRW, 2002], ambient calculus [CG, 1998] or graph rewriting [Rozenberg, 1997] are referred as suitable to model DCS but this specification method be easier and better to analyse.

In [BDTW, 2007] the Partner Abstraction approach is presented and Figure 7 illustrates the idea of combining nodes based on the similarity of their neighbourhood.
This abstraction takes all objects currently involved in the scenario into account. In contrast to this the Spotlight Abstraction ([BTW, 2007], [Toben, 2008] and [TWR, 2010]) is heterogeneous in the sense that the behaviour of a finite number of agents is preserved while the others are only abstractly represented (see Figure 8).

The abstraction techniques will be further investigated by Task 7.3 and therefore this section covers only an overview about this research area

### 1.2.4 UPDM (MoDAF, DoDAF)

#### 1.2.4.1 Introduction

UPDM is the Unified Profile for DoDAF (Department of Defense Architectural Framework, US) and MODAF (Ministry of Defence Architectural framework, UK). It is the OMG (Object Management Group) standard based upon UML (Unified Modelling Language), SysML (Systems Modeling Language) and SOAML (Service Orientated Architecture) that brings together, in a common metamodel, the key military architectural frameworks that are currently in use today.
1.2.4.2 The history of DoDAF, MODAF and NAF

An architectural framework is a structured of capturing the information relating to a particular domain. Among the first architectural frameworks, based upon Systems Engineering thinking, that was produced was the Zachman Framework [ZAC 1987, TOG 2006]. This showed a matrix that captured the relationships between the various questions that are asked when developing something, the “Who”, “What”, “Where”, “When”, “How” and “Why” mapped to the various stakeholder groups, the “Owner”, “Planner”, “Designer”, “Builder” and “Sub-Contractor”, involved with the thing being developed. The Zachman Framework was a loosely defined mapping that showed the sorts of diagrams that could be used to capture the relevant information at the intersection of any of the stakeholders and the questions that they wanted to be answered. The Zachman framework provides the basis for the types of diagrams, and the relationships between them, that are used in most of the architectural frameworks employed today.

In the United States during the 90s a number of frameworks evolved based upon the needs of the US government, the framework developed by the Department of Defence was originally called C4ISR (Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance)[DOD, 1997]. This took the Systems Engineering principles and the Zachman Framework and tried to formalise them into standard sets (Views) of diagram types (products). This evolved into DoDAF 1.0 during 2003 [DOD 2003]. At this time DoDAF 1.0 covered the following views.

- All View: Architecture summary documentation and data dictionary
- Operational View: High level organisational representation of the scenarios that need to be implemented in Systems
- Systems View: The system of systems that achieve the implementation of the operational requirements
- Technical View: Details standards that constrain the Operational and Systems views

The Operational and Systems views capture structural, behavioural, performance constraints and information aspects of the elements that they are representing.

DoDAF 1.0 was expanded 4 years later to include service (DoDAF 1.5 [DOD 2007]) so that they could capture aspects of Net-Centric warfare. Unfortunately this was not well though through and the way that services were defined was almost identical to systems.

Unfortunately during the early years of DoDAF, not may people understand the need for a common a information model (even though one was defined) and the concepts common to a model based approach. This lead to a lot of PowerPoint engineering that resulted in a lot of project overruns and bad integrations between systems. The Department of Defence did not fully understand what DoDAF was, neither did the main defence contractors.

During the mid 2000s the Ministry of Defence started to develop a more formal information model based upon UML and the concepts behind DoDAF 1.5. The result was MODAF 1.0/1.1, The NATO Architectural Framework was 90% identical to MODAF. The current version of MODAF is MODAF 1.2 [MOD 2012]. During the development of MODAF 1.2, the information model behind MODAF and NAF was synchronised. The main differences being some changes in view names (everything was prefixed by an N in NAF) and some minor refactoring of the diagrams used to capture data and information.

The viewpoints in MODAF and NAF expanded the original view set of DoDAF 1.5 to include

- Service Views: Stripped down set of essential views that capture Service taxonomy, behaviour and constraints
- Strategic View: Captures high level capability requirements, how they will be developed through enterprises and how they are related and deployed across organisations
- Acquisition Views: Relates the organisations that will develop the systems that are specified in the Systems views to projects, programs and timelines

It was this version of MODAF that provide the basis for the official version of UPDM 1.0 [UPDM, 2009].
Due to the issues that the DoD were seeing with how people were using DoDAF it was decided to create a formal information model for DoDAF based upon the IDEAS (International Defence Enterprise Architecture Specification) Ontology [IDEAS, 2009]. IDEAS is what as known as a 4 D Ontology, it is set based and it is firmly rooted in things that exist, that have temporal and spatial extent, i.e. individuals that exist in space and time. The identification of the elements that exist in the IDEAS foundation is based upon the BORO Methodology [PART 1996]. It is worth noting that many of the concepts that exist in MODAF are based upon IDEAS as the people behind MODAF were key people in the development of IDEAS. The result of this effort was DoDAF 2.0 [DOD, 2009a][DOD, 2009b].

DoDAF 2.0 took a lot of the concepts in MODAF and NAF and renamed them
- Strategic views became Capability Views
- Acquisition Views became Program Views
- Data and Information views from NAF became the DIV views in DoDAF 2.0

Despite this there is still an underlying set of core concepts that relate all these three frameworks together. The Unified Profile for DoDAF and MODAF V 2.0 [UPDM, 2012], [BLEAK, 2010] is the means by which the information model for these frameworks (MODAF 1.2004, DoDAF 2.0.2 and NAF 2.1) has been brought together and standardised in a UML, SOAML and SysML profile.

1.2.4.3 The Unified Profile for DoDAF and MODAF (UPDM)

UPDM is a profile, based upon UML, SysML and SOAML. It described the mappings and constraints that exist between the elements that exist in the frameworks and the elements of UML, SysML and SOAML. The mapping is based upon the definition of a domain metamodel that captures the semantic of the elements and the relationships that exist between them. This was then turned into an implementation model that showed how the elements were mapped to implementations in the various languages. This was divided up into a set of core concepts and then extend across the views and viewpoints to capture specific MODAF and DoDAF aspects of the frameworks. Sets of diagrams and the elements that should appear on each of those diagrams was also specified.

The diagrams shown as part of section 3.2.1 were created based upon UPDM and show, in an example based upon Search and Rescue, how the UPDM information model can be used to capture architecture information in the relevant DoDAF/MODAF diagrams.

1.2.4.4 The rationale behind UPDM

UPDM was developed for a number of reasons:

1/ Governments are becoming more reliant on commercial of the shelf standards. They are starting to realise that creating, publicising, marketing and more importantly maintaining these standards is becoming very costly. Governments want to reduce these costs and get more tool vendors, government and industrial organisations to use them properly

2/ During the early years of DoDAF and MODAF many of the users were guilty of what is know as Powerpoint engineering. The diagrams that they were producing were inconsistent in the information that showed across the diagrams and in some cases within the diagrams themselves. UPDM relies upon a model based approach to architecture where a change in one element that is used on several diagrams can be seen across all those diagrams. Also the tools used to develop UPDM architectures are also developing consistency checkers to ensure that the models produced are structurally sound.

3/ There is plethora of Architectural frameworks in current use across the globe. Although many are based upon the core concepts of DoDAF, many are not. This has led to a tower of babel of AF
languages and terms. UPDM provides a common understanding of the elements and relationships within the frameworks that it supports.

Prior to UPDM, each of the tool vendors had developed their own proprietary profile based upon the various handbooks and definitions that existed for each of the frameworks. The result of this was that none of the tools could talk to each other. One of the key requirements of UPDM was to show interchange of models between tools. This lead to the formation of the Model Interchange Working Group in the OMG who have been steadily progressing to show interchange between the various UML tools based upon XMI. Now that foundation work on UML and SysML have been completed UPDM is next on the list.

## 1.2.4.5 UPDM 2.0 Compliance Levels

There are two compliance levels in UPDM, signified by the terms L0 and L1, these terms are used to indicate the variation in OMG profiles that are used to support UPDM.

![](image)

**Figure 9 UPDM Compliance levels L0 and L1**

- **L0**: This compliance level is based on UML and imports the service and capability related elements of SOAML. The L0 levels also reuses a minimum of set of SysML elements including Requirements, Views, and Viewpoints.

- **L1**: This compliance level contains UPDM L0 and imports the entire SysML profile, although only part of it is referenced by UPDM element. This compliance level contains a set of constraints specifying the SysML stereotypes to be applied to the L0 elements. This compliance level provides integration with system modelling using SysML.

The benefit of using L1 for is that it provides a semantic correctness between certain UPDM elements SysML concepts, e.g. Systems are extensions of SysML blocks, Flowports are used to capture data flow interfaces. The concepts of requirements does not exist in UML, Allows you to capture constraints as parametric diagrams and, depending upon the tool, execute them. Helps with the creation of executable models. Many of the SysML/UML tools allow users to create executable behavioural models. This is of real interests to Operational and Systems modellers as they need to verify their understanding of these models and validate them against the original customers requirements. Keeps a consistent modelling notation language from high
level capability definition down to software implementation. One of the big drivers for UPDM came from the Defence industry (big users of SysML and UML based tools) who wanted a way to take Systems of Systems models down to Systems Models and finally UML models using the same underlying language. Because UPDM is based upon SysML and UML it is possible to do this, without losing information.

1.2.4.6 DOD/MOD support for UPDM and international standardisation

The DoD and MOD have both been supportive of the efforts that have gone on in the UPDM group, physically through making available expertise in the fields of DoDAF and MODAF and morally by keeping faith with the direction where the UPDM group is going.

The DOD has mandated UPDM 1.0 (DoDAF 1.0/1.5) and UPDM 2.0 (DoDAF 2.0) for the development of military architectures if using UML and/or SysML tools.

UPDM 1.0 is going forward as an ISO standard, and once UPDM 2.1 has been completed this will also be put to the ISO for consideration as ISO standard.

1.2.5 Multi Agents

MESSAGE: a multi-agent systems engineering methodology: The MESSAGE methodology is based on a set of core concepts: the agents, the agent organization, the role allocation, the resources, the goals, the tasks and the functional workflows. Using these core concepts and a set of refining activities, the methodology tries to improve the design of multi-agent systems by considering the design of such systems as a successive top-down and bottom-up steps.

1.2.5.1 The main concepts definition

1. The agents are the main entities: An Agent is an autonomous entity able to perform functions and to deliver services. The quality of “autonomy” means that an agent’s actions are not entirely designed through external events or interactions, but also by its own motivations and goals.

2. The organizations: Agents are put inside a hierarchical organization that fixes the main relationships between them such as superior-subordinate and customer-provider, etc. In addition to the agents/entities organization, other organizations such as task decomposition are considered.

3. The role allocation: Role concept is very important, because it allows separating the contextual responsibilities of an agent from its identity and general behaviors. A Role describes the external characteristics of an Agent in a particular context. As an example, an Agent may be capable of playing several roles, and multiple Agents may be able to play the same Role. Then “Roles” can be viewed as reusable patterns and as a generic way of behaving.

4. The resources: Resource is used to represent external constraints, means used by the agents such as databases, external programs, production means…

Once the previous main concepts are defined and organized in a first way, then the methodology focuses on analyzing the tasks/goals imbrication. The objective is to decompose the tasks/functions that should be delivered by the system and to allocate the goals to the agents that will achieve these functions.
1.2.5.2 The methodology steps

The first analysis and the core concept definition enable to decompose a problem following two main views:
- the first view enables to analyze the problem regarding the component/organic decomposition;
- the second view is used to analyze the functional behaviors that the system should deliver.

By coupling these two views, a first refinement analysis enables to build connections between the agents and the activities to be done. At the end of this coupling, agents have a set of roles, the roles are associated with goals and tasks, and tasks are organized in workflows that describe how to perform main system scenario and how to achieve the main system behaviors.

The methodology is then based on an iterative refinement approach: the system is viewed as a set of organizations that interact with resources, actors, or other organizations. Actors may be human users or other existing agents. Subsequent stages of refinement result in the creation of models at level 1, level 2 and so on. At level 0, the modeling process starts building the Organization and the Goal/Task views. Finally the Interaction view is built using input from the other models. The level 0 model gives an overall view of the system, its environment, and its global functionality. The granularity of level 0 focuses on the identification of entities, and their relationships. More details about the internal structure and the behavior of these entities are progressively added in the next levels. In level 1 the structure and the behavior of the organization, agents, tasks, goals and domain entities are defined. Additional levels might be defined for analyzing specific aspects of the system dealing with functional requirements and non-functional requirements such as performance, distribution, fault tolerance, security. There must be consistency between subsequent levels.
1.2.5.3 Synthesis and added value for SoS (vs.UPDM)

The views consistency: The experience in MESSAGE shows that the different views of the system help the analyst to choose the most appropriate strategy. In practice a combination of refinement strategies with frequent loop-backs among them are used. The analysis process might start with the Organisational View, then switch to the Agent View and continue with the Interaction View. The results of the analysis of specific interaction scenarios may lead to reconsider part of Organisational View, and thus to further refining and adapting of Organisation visual constituents. But the methodology doesn’t provide any formal process or any formal model to be considered.

Definitions from MESSAGE methodology:
- Agent/Role view (AV) – This focuses on the individual Agents and Roles. For each agent/role it uses schemata supported by diagrams to its characteristics such as what Goals it is responsible for, what events it needs to sense, what resources it controls, what Tasks it knows how to perform, ‘behaviour rules’, etc.
- Interaction view (IV) – For each interaction among agents/roles, this views shows the initiator, the collaborators, the motivator, the relevant information supplied/achieved by each participant, the events that trigger the interaction, and other relevant effects of the interaction (e.g. an agent becoming responsible for a new goal). Larger chains of interaction across the system (e.g. corresponding to uses cases) can also be considered.

![Diagrams of different views of the system]

Figure 2: Different views of the system

In addition to the iterations on the different views, the tradeoffs and the iterative analysis are also performed between bottom-up and top-down considerations. This is a result of the multi-agent properties. Because each entity of these systems is considered as partially autonomous and independent, it is important to consider that parts of the system properties are emergent and have to be validated/tested/adapted through simulation tradeoffs.
1.2.5.4 KAOS: A goal oriented methodology

The KAOS methodology originated from a cooperation between the University of Oregon and the University of Louvain, Belgium, in 1990. It has been further improved by the University of Louvain. KAOS features different models: goal modeling, responsibility modeling, object modeling and operation modeling. The commercial Objectiver tool offers support for the KAOS methodology. Objectiver is being developed and distributed by Respect-IT, a spin-off company from the University of Louvain. [CESAR, 2011]

Figure 3: An overview of KAOS concepts

1.2.5.4.1 Main concepts

KAOS is also a multi-agent inspired methodology, but it is focused on the goal modeling. This goal modeling involves the following concepts:

- **Goal**: A goal is used according to goal-oriented requirements engineering. It may be refined by other goals, domain properties, expectations and requirements. According to the methodology, a refinement shall answer the question how a certain goal can be achieved. Traversing the same link towards a more abstract goal shall clarify why a certain requirement, goal etc. has been expressed. Goals can be declared as being conflicting with other goals. A special type of goal is the soft goal. The root goals, from which all other goals are directly or indirectly derived by refinement, describe the top-most strategic purposes of the whole system. The KAOS meta-model makes possible to refine a goal by several sets of sub-goals in order to start an alternative analysis. Different tactics can be used for goal refinement: case-driven decomposition, milestone-driven decomposition…

- **Requirement**: A requirement is a low-level goal derived from a higher-level goal and assigned to a certain software agent, which is responsible for fulfilling the given requirement. This means that when a software system is being implemented, a given set of requirements has to be fulfilled. A requirement is typically not further refined (unless all its sub-requirements are assigned to the same agent as the one responsible for this requirement).

- **Expectation**: The goal type of an expectation is similar to a requirement, except that it cannot be implemented or designed. An expectation is an assumption about an agent in the
environment of the system-to-be-developed. This, for instance, might involve responsibilities which are assigned to a pilot, driver or train operator.

- **Domain property**: A domain property is a descriptive assertion regarding the application domain. It may be a domain invariant (something which always applies in a certain domain, some standard for instance) or a domain hypothesis about some object to hold. Domain properties can be used to show that some refinement is complete.

- **Obstacle**: These are used to model situations, where a goal, requirement or expectation might not be met due to some undesired behavior. In that case, an obstacle is said to obstruct one of the other elements. In the same way as goals, obstacles can be refined in sub-obstacles if needed. New requirements can be added to address each leaf-obstacle (this relationship is then represented by a resolution link).

### 1.2.5.4.2 Main views and models

The previous core concepts are included in four key models:

- **Goal modeling**: A goal might be refined towards sub-goals, requirements or expectations. The refinement serves as a possibility to introduce different levels of abstractions as well as delivering justifications for sub-goals or requirements.

- **Responsibility modeling** is close to the role association in MESSAGE methodology. In this view/model, the objective is to associate goals/requirements to agents. Agents are then responsible to achieve goals in a defined system environment. Assigning responsibility towards an agent shall clarify who is expected to fulfill a certain requirement or expectation.

- **Operation modeling**: Once the goals are decomposed and allocated to agents, the system operations have to be defined. It means that once the roles are allocated to the different entities, we have to define how these entities collaborate together to achieve the system goals. In these models, KAOS builds the reasoning about when is what to be done and how it is to be interfaced between entities (serving as input/output). Events are also identified as operations to be triggered.

- **Object modeling**: The last modeling activity enables to build the view on the domain concepts. The objective of this view is to build the meta-model of the domain and to link it to the operation and responsibility modeling in order to specify properly what the manipulated concepts are. For example, this model can be used to enrich/describe the interfaces of an operation and used furthermore to enrich the description of a goal in a sense that it expresses which domain concepts are concerned by a certain goal…

As represented in Figure 3, these views are partially overlapping so that the requirements engineer can choose where he wants to add the information first and the other views can then be automatically updated. The steps to follow as guidelines to build the whole KAOS model are described in the next section.

### 1.2.5.4.3 Typical workflow

Although the process of building a KAOS model is generally not linear (it is possible to go back and forth whenever new elements are identified), here are the ten key ideas underlying the methodology [Respect-IT, 2007]:

a. **First build a requirements model**
   - A mix of top-down and bottom-up approach is recommended. By interviewing stakeholders, intermediate goals are usually identified first, from which higher level goals are then retrieved and requirements added via refinement.

b. **Justify the requirements by linking them to higher-level goals**
   - Each goal introduced in the model must be justified by at least another higher-level goal (except for the top-most strategic roots).

c. **Build a model of the system + the part of its environment that is involved in the interactions**
   - Requirements on agents interacting with the system to be designed are known as expectations in the goal model. They are introduced to show how the system and its environment have to cooperate to achieve goals of the system. Assumptions about the environment are also to be recorded.
d. Build the responsibility model
   - Each requirement must be associated to a single system agent, who becomes responsible for it. Unlike requirements, expectations are achieved by an agent belonging to the system’s environment.

e. Build a consistent and complete glossary of all the problem-related terms used to write the requirements
   - This is important to make sure that all stakeholders share the same definition of the concepts used.

f. Detail how the agents need to behave in order to satisfy the requirements they are responsible for
   - Add an operation to each pair of requirement/agent (the agent responsible for the requirement performs the operation that operationalizes this requirement). The operation diagrams, composed of operations linked together in terms of input/output (event or dataflow) and cause (event), must also be completed to depict the cooperation needed for achieving the requirements.

g. Base the requirements document on the requirements model
   - The requirements document can be automatically generated from the model.

h. Validate your requirements by first reviewing the model
   - The KAOS model reveals to be a convenient mean to communicate about the requirements, bridging the two worlds of analysts and users.

i. Use a defensive approach to the building of a requirements model
   - It is a good practice to try and challenge the requirements by identifying potential obstacles, which in turn can be resolved by finding new requirements.

j. Consider your requirements document as a reference that shall need updating during the project development life cycle
   - It is by no way static. If needed, the requirements document can be regenerated automatically each time an update is made in the model. A model is usually much easier to maintain than a document.

Four conditions have to be fulfilled for the model to be complete:
- every leaf goal is either a requirement, an expectation or a domain property
- every requirement is placed under the responsibility of one and only one agent
- a process diagram must specify the agents who perform the operations and the input/output data for each operation
- a process diagram must specify when operations are to be executed (i.e. which events can trigger the operations)

1.2.5.4.4 Synthesis and added value for SoS (vs.UPDM)

The added values of the KAOS methodology are similar to the MESSAGE’s one. First, we saw that the consistency between views were of first importance in MESSAGE, even if the way to proceed between the different views wasn’t clearly described. In KAOS, this issue of consistency between the different modeling activities is also important and seems clarified step by step along the tradeoffs. In addition, the KAOS methodology seems to use object modeling activities as a support for the consistency between other activities.

See also:
- Modeling tools

Objectiver for KAOS - [http://www.objectiver.com](http://www.objectiver.com)

Agent methodologies

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Date</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>Draft</td>
<td>2012-06-04</td>
<td>23 of 57</td>
</tr>
</tbody>
</table>
Requirements collected from end-users in WP3 provide some insight about “where to go” regarding Goals and Contract Specification Language for SoS – possibly in relation with SoS architectural models. This is summarized below. Please refer to D3.1 for more precise explanations and illustrations of user needs. A general expectation is to make explicit the relations between “contracts” on one hand, “requirements/properties” on the other hand. This calls of course to some extent for methodology support, but in the context of WP6 call for a formal expression of goals, especially quantified goals – in relation with requirements which are not necessarily formal in the mathematical sense, and properties attached to models of systems and components (as opposed to “emerging” properties, which cannot be attached to a specific system or component).

According to end-users perspective, SoS achieve their goals by relying on “constituent systems”, where each constituent system can itself be a very complex system. For example, the European Air Traffic Control is a SoS which relies on Air Traffic Control Centres, Airports, Airlines, Aircrafts, Navigation equipments, and so on. Each constituent system must support the overall goals of Air Traffic Control, such as maximizing traffic while minimizing energy consumption and environmental impact – and, of course, avoiding accidents. Constituent systems may have local goals, related to managerial independence, which implies the need for collaborative decision making of human authorities in some cases (e.g., crisis situations). Local goals are not necessarily aligned in all situations with global goals.

In addition, according to the evolution timescale of such SoS, constituent systems may not provide the ideal capabilities to support global goals, which can be described as a “legacy handling” issue. For example, the future Air Traffic Management will handle “4D trajectories” management, which needs support from a new type of Flight Management Systems onboard aircrafts. It is expected that the DANSE solutions will address the issue of non harmonized support of global goals.

Taking as example Water Management SoS, it is clear that legacy systems may have local goals, which may contradict some overall SoS goals. For example, a local system may have a goal of increasing revenues while the overall SoS goal is minimizing cost to the public. The DANSE work products shall enable the goals mapping of both the SoS and the constituent systems.

The process to map global goals at SoS level either on services provided by constituent systems, or to refine global goals at SoS level as local goals for constituent systems should be formalized, consistently with the architectural representations at each level.

In the design activities, support is expected for the mapping of goals. In the verification activities, once goals have been mapped, it should be possible to identify whether goals are aligned, contradicting or indifferent; a weighted assessment on the overall alignment (or lack of) should be provided.

Considering the implementation of tool support for goals and contracts specification languages, it is foreseen that the solutions proposed by DANSE will have to fit into some existing methods and tools, globally referred to as “UPDM”. An expectation is that the set of formal representations supporting SoS engineering should express the main stable concepts underlying NAF/DODAF/MODAF de facto standards, but should not be directly dependent on a particular version/variant of these standards. These formal representations should be robust to medium term foreseen evolutions of UPDM, and in particular should not be dependent on SySML compatibility issues.

Goals and contracts are expected to be different from, but related to the (required or provided) properties attached to architectural models. Considering implementation issues, the SoS metamodel and the goals and specification languages should consider ongoing evolutions of OMG standards; this include in particular, the
ability to capture non functional properties using MARTE annotations on architecture models, and the
discrete time model defined by MARTE.

1.4 Specific requirements

The requirements are taken from D3.1 and are identified by their respective reference numbers. The
requirements database is a “living” document constantly updated as the project progresses. Therefore, for
the most current version of requirements, refer to the requirements database. The reference numbers for the
requirements will remain constant.

Requirements may be articulated as direct requirements from the SoS Modeling Formalisms, or as a direct
requirement from another aspect of the DANSE framework (i.e. another WP) with an indirect impact on the
SoS modeling. The dependency analysis has been done in the scope of WP3.

1.4.1 RQ-05-004

Requirement: The DANSE toolnet shall provide a process to abstract the model of component systems
(defined with detailed models) to be integrated into the SoS model without losing the fidelity in the
representation of the functionality or behavior of the constituent system, thus affecting the fidelity of the SoS
model.

Discussion: Since the constituent systems are modeled in a more detailed (refined) manner than the SoS
framework requires (or can handle), there is a need to be able to extract the information from those models
at the required abstraction level. A convention is needed to define what model properties are to be used on
the SoS modeling level.

1.4.2 RQ-05-011

Requirement: Provide a methodology for verification of SoS

Discussion: In any verification effort a set of goals and a set of verification criteria must be defined. Correct
SoS behavior must be specified in a formal, computer readable way to enable the automated tools to
compare these specifications with the actual model behavior.

1.4.3 RQ-10-007

Requirement: Abstractions techniques and metamodel shall support combination of physical views of system
architecture (e.g., relying on stable environmental and geographic data, at the level of detail appropriate for
the considered SoS) as well as high-level stable set of evolution rules and service orchestration principles,
abstracting irrelevant considerations of system implementation.

Discussion: Tightly related to RQ-05-004. This requirement emphasizes the techniques rather than the tool
that implements them. The metamodel must allow the depiction of the SoS architecture including the
modeling of all constituent systems (at the required abstraction level) and the relations between them,
whatever the nature of those relations can be.

1.4.4 RQ-05-024 & RQ-05-025

Requirement A: The tool net shall support SOS-level, cross-tool, distributed modeling.
Requirement B: The tool net shall support cross-system, cross-tool, cross notation distributed modeling with
appropriate constituent system model abstraction without losing meaningful info.

Discussion: The SoS model must be able to include information about the individual model (a component of
SoS) source and its various parameters that may affect the way the model is handled in simulation,
verification, optimization or in any other type of processing. The model information must also include non-
functional data such as controlling entity, model version, confidentiality parameters and other information that
may be needed for model management in a multi-user, multi-tool distributed environment.
1.4.5 RQ-05-006 & RQ-05-028

Requirement: The DANSE work products shall enable the goals mapping of both the SoS and the constituent systems.

Requirement: The tool net shall support cross-system, cross-tool constituent systems local goals alignment with SOS global goals.

Discussion: Since SoS makes the distinction between SoS goals and the individual players (constituent systems) goals, a modeling method must be provided to separate the two and furthermore to provide linking and tracing ability from one set to the other. The linking ability should include information related to the link to enable capturing various relations between the SoS and the individual players.

1.4.6 RQ-10-002

Requirement: For large and highly dynamic systems, the concept of "system mode" may make sense at the level of operational analysis, but does not easily translate in terms of "states and modes" mapped on system architecture. The SoS metamodel and the methodologies for handling dynamicity should therefore propose a means to describe system evolution using a statically defined set of evolution rules, as a complement to the usual description in terms of transitions between a statically defined set of states.

Discussion: A method is needed to define a SoS mode, as one of the possible operating configurations of the SoS. Also the transition between the SoS modes need to be defined (triggers, actions, etc). The mode definition is needed for all types of analysis.

1.4.7 RQ-10-003

Requirement: The set of formal representations supporting SoS engineering should express the main stable concepts underlying NAF/DODAF/MODAF de facto standards, but should not be directly dependent on a particular version/variant of these standards. These formal representations should be robust to medium term foreseen evolutions of UPDM, and in particular should not be dependent on SySML compatibility issues.

Discussion: Same as SoS the modeling formalisms are in a constant state of flux. The DANSE methodologies must be able to cope with the changing standards, to remain current and useful. Thus, since the medium term evolution of Architecture Frameworks is known, effort must be made to ensure that changes in the standards will not break the DANSE framework.

The set of formal representations supporting SoS engineering should express the main stable concepts underlying NAF/DODAF/MODAF de facto standards, but should not be directly dependent on a particular version/variant of these standards. These formal representations should be robust to medium term foreseen evolutions of UPDM, and in particular should not be dependent on SySML compatibility issues.

1.4.8 RQ-05-001

Requirement: The DANSE work products shall enable the creation of a SoS model based on constituents systems' models that were created with legacy tools.

Discussion: Related to RQ-05-24/25. The formalism must be able to enrich the legacy models with enough information that the participation of the legacy models in the tool net is possible.

1.4.9 RQ-10-009

Requirement: The SoS metamodel should consider ongoing evolutions of OMG standards; this include in particular, the ability to capture nonfunctional properties using MARTE annotations on architecture models, and the discrete time model defined by MARTE.
2 Proposal for a comprehensive DANSE approach for SoS

This Section overviews a proposal for a comprehensive DANSE approach for SoS. The DANSE consortium has identified five dimensions that are considered essential from the designer’s point of view. These are:

- Operational independence
- Managerial independence
- Evolutionary development
- Emergent behaviour
- Geographic distribution

This section shows how techniques proposed in Section 2 could be used/improved to fulfill those requirements.

2.1 Test and Evaluation Solutions

Test and evaluation is focused on the operational view. The test and execution can be carried out when the operational view includes operational activity diagrams and operational state transition diagrams (see Figure) for each activity. Operational state transition diagrams are required to make the activity diagram executable.

![Example of operational state transition diagram](image)

Figure: Example of operational state transition diagram

Test and evaluation techniques for the operational views can take advantage of test generation methods developed in the scope of UML activity diagrams for software testing.
In the authors propose an approach to generate test cases from design models using activity diagram, sequence diagram and class diagram. Their approach parses the activity diagram and generates the test scenarios which satisfy the path coverage criteria. As operational activity diagrams in UPDM may not represent an executable implementation, this method must be extended to also include operational state transition diagrams, in order to identify possible executing paths.

In the authors propose a test case generation methodology that augments the activity diagrams with necessary test information, then converts the activity diagram into an activity graph representation, which automatic test case generation techniques can be applied on. As in the previous case, the method must be extended to also include operational state transition diagrams.

In the authors propose the methodology depicted in Figure: the methodology constructs an intermediate table called the Activity Dependency Table (ADT). The produced ADT table automatically generates a directed graph called Activity Dependency Graph (ADG). The ADG is then examined using the Depth First Search (DFS) in order to extract all the possible test cases. The ADT’s form makes the ADG cover all the functionalities in the activity diagram. The generated test cases should go through all the branches in the activity diagram. Thus, it applies the branch coverage criterion. They also justify all the conditions/predicates that might be established in the activity diagram as well as all the basic paths (which are paths that cover loops zero or one time) because the model checks all the loops only once. The generated test cases are validated against the Cyclomatic complexity technique to check all the test cases generated by that technique. Thus, the authors claim that the Cyclomatic complexity coverage criterion has been verified as well.

In the authors describe a Java/eclipse based platform called PETA for automated software testing based on UML activity diagrams. The test case generation flow described is as follows:

- Draw the Activity diagram representing the component;
- Input file is created corresponding the Activity Diagram which is in the form [initial state, transition, next state];
- Store the input file into appropriate data structure, e.g. hash map;
- Create the final state transition table, adding new states to handle fork and join;
- Finally test scenarios are generated.

In the authors present a test case generation three-step methodology based on test suite generation, concretization and evaluation:

- Test suite generation: Following industrial practice, we use activity diagrams to express low level requirements. In an automatic step, we translate activity diagrams into fragmentary C code containing control flow statements, stubs, and labels;
• Test suite concretization: Each of the test cases generated in the previous step is realized as a concrete test case on the implementation. In order to fulfill the black box criterion, we need to verify that there is a 1-1 correspondence between the test cases from the activity diagrams and the test cases on the implementation. At this point, the following two deficiencies in the model-implementation slope can occur:
  o Implementation poverty: If some test case cannot be concretized, then the model contains behavior not implemented in source code. The reason can be either on the side of an incomplete implementation, but also in erroneous or imprecisely formulated requirements. Note that implementation poverty can be spurious, if the model does not reflect relevant dependencies of the involved control flow decisions;
  o Implementation liberty: The implementation contains more than one concrete control flow path for some abstract test case. This situation expresses that the activity diagram is an abstraction of the implementation, and thus, the model does not fully determine the control flow of the implementation;

If neither deficiency occurs, then for each abstract test case there exists exactly one control flow path to be captured by an implementation test case which matches the abstract test case, and vice versa;

• Test suite evaluation: In this phase, the test suite obtained from the previous steps is executed on the implementation. It should achieve MC/DC (C3) and run on the implementation without errors (C4). Two further deficiencies may occur in this phase:
  o Implementation anarchy: The implementation contains behavior not covered by the test cases and therefore not contained in the model. Reasons for this situation include programming errors, reuse of code, use of COTS components, or undocumented functionalities;
  o Implementation error: Either code assertions are violated or some program output is inconsistent with the requirements;

If neither deficiency occurs, then the test suite covers the implementation and does not reveal any disagreement with its requirements.

In the authors present a test case generation based on the condition slicing technique. Their objective is to minimize test cases that cover the maximum paths. It is possible by slicing the activity diagram on the basis of applying pre-conditions on control nodes. In their approach, the authors first select a conditional predicate on an activity edge during the execution of an activity diagram with a random input data set. Subsequently, they are checking each path in the flow dependency graph by slicing the path and checking the simplicity of the path.

In the authors combine activity diagrams and statechart diagrams to automatically generate test cases and to this purpose they introduce and formally define state activity diagrams (SAD). The authors propose a test case generation technique that is based on the interactions among the objects. This test data can be applied during the integration test phase, right after the completion of class testing. They have named their proposed approach SATEC (State Activity TEst Case generation). The methodology is depicted in Figure .

![Figure: Test case generation methodology based on state activity diagrams (SAD)](image-url)
In the authors combine activity diagrams and sequence diagrams to generate test cases. Using sequence diagram, it is only possible to show message paths. But by using sequence and activity diagrams it is possible to cover message as well as activity path which is called message-activity-path. So errors uncovered in message-activity-path cannot be uncovered by message-path. But the reverse is possible. Thus message-activity-path coverage which is the super set ensures message-path coverage which is subset. The test generation process is divided into three main phases. The first phase is to generate the Model Flow Graph (MFG) internal representation from sequence and activity diagram separately. The second phase is to generate test sequences from MFG corresponding to sequence and activity diagrams. The test sequences are a set of theoretical paths starting from initialization to end, while taking conditions (pre-condition and post-condition) into consideration. Each generated test sequence corresponds to a particular scenario of the considered use case. The third phase is to generate test case from the generated sequences satisfying the message-activity path test adequacy criteria.

In the authors present a test case generation technique based on statistical usage models. Statistical usage testing is based on the idea that different parts of a program do not need to be tested with the same thoroughness. There is the (infamous) 90-10 rule stating that typical software spends 90% of the time executing 10% of the code. That means, different portions of software are executed with a higher frequency than others. Statistical usage testing aims at identifying these portions and adjust test suites accordingly, subjecting more frequently executed sections of code to more thorough testing. The authors show a methodology to build the statistical usage models: starting from refined use case diagrams, they build state diagrams, from which usage graphs are derived, which in turn are refined into usage models, from which test cases are generated.

2.2 Graph-Transformation Systems

Our presentation does not use these five dimensions as a structure, since the logics of the development proceeds much more comfortably with a different document structure. Therefore, the five dimensions are discussed in a concluding section, in the perspective of the presented approach.

2.2.1 Some considerations regarding UPDM

DANSE sees UPDM as a background. UPDM offers the advantage of well structuring SoS specification by focusing on top-level. However, like most formalisms developed as standards, UPDM is overly rich in expression means and offers probably too many, often overlapping, views. We wish to clean UPDM from its unnecessary views (there are many of them) and recommend some missing views that, as we believe, are essential. We may need to enrich UPDM with a limited number of new concepts and “tools” (or “views”) to equip UPDM models with clean semantics. The next paragraphs provide elements to help understanding the essentials of UPDM by abstracting away from some ontological details.

2.2.1.1 Capabilities

Capabilities are a central concept in UPDM. A capability is something that the overall SoS can offer.
Figure 10 shows an example of using capabilities as a SySML extension to support requirement capture. Capabilities are listed on the boxes sitting on the bottom and the diagram on the upper part of the diagram describes what the capability is supposed to offer as a list of properties. Note that such properties can be turned into goals by giving them a quantitative reward telling how good they are fulfilled.

Some further examples of capabilities are shown on Figure 11.
Discussion: There is a need to clarify capabilities versus goals. Both concepts are indeed overlapping. According to some opinions, goal is meant of higher level and global to the SoS. Goals may be phrased in informal terms. In contrast, capabilities are actual properties that result from the delegation of part of the goals to some services (or orchestrations thereof) or systems or sub-SoS.

To clarify the matter, it is proposed to focus on the following concept of goal. A goal is:

- a capability,
- enhanced with a metric telling how good the capability is achieved.

The metric is optional. It is meant to be a “distance to achievement” or a “probability of achievement”.

### 2.2.1.2 Services

Figure 12 shows examples of views combining goals (in green), services (in magenta), and interfaces (in blue).
Figure 12
2.2.1.3 Operational views

Figure 13 shows an operational view. Such operational views are complemented by two kinds of information. First, using activities, it is specified how services are orchestrated, see Figure 14.

Second, the mapping of services to systems, seen as resources, is specified, see Figure 15.
2.2.1.4 System views

System views are provided in the form of architecture descriptions, see Figure 16.

Such views are complemented with rules driving the interaction between the different agents, see Figure 17.
A very important diagram related to system view is shown on Figure 18. This figure shows a Gantt diagram, where time is the horizontal dimension and the different activities or services are triggered by this time. Such Gantt charts are the only mean offered by UPDM to specify how the SoS can be reconfigured dynamically. This is a very crude mean to specify dynamic evolutions of SoS architecture. Means are missing that would allow for event-based evolutions. Indeed, event-based evolutions should be the basic mean offered, with time-based as a derived mean – dates are just one special kind of event.
2.2.1.5 Discussion

There are many different entities in UPDM that concur to the achievement of the SoS goals, namely: systems, services, organizations, etc. and even capabilities/goals. All these entities are related to one way or another; corresponding relations may need to get typed. It is proposed to encompass the above concepts through a general notion of *graph* \((G)\).

The advantage is that we factorize in this way several high-level operations on SoS at architecture level, such as:

- Initial Configuration of SoS;
- Deployment of new entity (goal, service, system, infrastructure);
- Deployment of new instances of existing entities;
- Re-configuration to handle alarms, volunteered by the command-and-control center;
- Add or modify goals.

*Graph Grammars (GG)* will provide us with the needed mathematical framework to support event-based dynamic evolutions of SoS architectures seen as graphs.

2.2.1.6 UPDM Gaps vs. specific requirements

The below list refers to 1.4.

RQ-05-004: UPDM does provide capability to link an abstract model at the structural/capability view to a refined model of a constituent system with detailed functionality. The extension mechanism provides an easy way to add information to the model pertaining the exact nature of the detailed model for use in an automated analysis.

RQ-05-011: UPDM does not include definition of goals or criteria for any type of analysis. However, such a concept may be added using the standard UML extension mechanism.

RQ-10-007: UPDM allows linking of structure views to the operations views. Coupled with the ability to define evolutions, UPDM should be sufficient to satisfy this requirement. Should the expressiveness of UPDM not be enough to capture all the architectural relations, an extension may be provided.

RQ-05-024/5: No such mechanism exists in UPDM. However, some of the DANSE consortium partners have experience in solving such problems in other projects by creating a language extension for adding the cross-tool information to the model [DARPA, 2011].

RQ-05-006/28: This requirement is covered by the ability of UPDM to link capabilities to operational activities. The capabilities in UPDM are the high level goals of the SoS and the operational activities are usually the functions of the constituent system.

RQ-10-002: SV-8 view of UPDM was created for the purpose of capturing the evolution axis of a system, which should be adequate for DANSE. The “system mode” concept can utilize the same evolution engine or, alternatively, extend the language to include definition of modes. Modes can range from simple on/off (exist/not exist) and to a more complex modes where multiple properties of a constituent system are affected by the mode switch.

RQ-10-003: An alignment with the evolution of standards in the SoS field is closely monitors. The DANSE consortium, thru IBM chief solution architects who participate in the relevant industry and government committees, will be able to foresee the future developments and adjust its roadmap and deliverables accordingly.
2.2.2 Graph Grammars (GG), a framework for dynamic SoS architectures

2.2.2.1 Principles and overall objectives

We need a mathematical framework supporting both reused features from systems and features that are specific to SoS. These are:

- Reused features from systems:
  - Systems;
  - Interfaces & Contracts as high-level abstractions of systems;
  - Services and Service Orchestrations; services are stateless activities that can thus be shared by many clients with no interference.

- Novel features specific to SoS:
  - Architectures supporting dynamicity and offering links to system level;
  - Goals and emerging behaviors.

To support the above features, we propose in Table 1 our mathematical frameworks and tools.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamicity</td>
<td>Graph Grammar (GG) with its transformation rules</td>
</tr>
<tr>
<td>Link between SoS and system levels</td>
<td>Guards involving system interfaces</td>
</tr>
<tr>
<td>Goals</td>
<td>Properties + Costs</td>
</tr>
<tr>
<td>Systems</td>
<td>Interfaces &amp; Contracts</td>
</tr>
<tr>
<td>GG transformation rules</td>
<td>Contracts</td>
</tr>
</tbody>
</table>

Table 1

Architectures and architecture elements are captured by graphs. The framework of Graph Grammars (GG) that we recall next will formally capture the dynamic evolution of architectures. More precisely, GG rules will be used to specify the “constitution” of SoS architectures, i.e., the set of legal dynamic evolutions. Link between top-level SoS architecture description and system-level description is seamless offered by guards. Guards disable or enable GG rules, thus triggering some evolutions among the legal ones in a given SoS configuration. Guards involve parameters, ports, and variables from the System Interfaces, thus formally establishing the link between SoS- and system-level. Goals consist of a property together with a metric assessing how good it is satisfied. Interfaces and contracts can be used to specify abstractions of systems and GG rules – the latter are statically specified by the SoS designer.

2.2.2.2 Sketching an illustrative example: the “rescue” SoS

To illustrate our purpose we consider a very simplified version of the rescue case study with its following list of entities:

- Organizations (superstructures)
  - Media
  - Army
Civil authorities

Systems (infrastructures)
- Social networks
- Hospitals
- Electricity Grid
- Road
- Train
- Air
- Telecom Networks

Services
- Broadcast info
- Electricity
- Transport
- Communication
- Emergency services

Capabilities (goals)
- Return to normal state (quantified?)
- Avoid worsening the situation
- Minimize the use of resources specific to this incident
- Availability of infrastructure
- Availability of coordination
- Post-incident analysis

Strategies and Policies
- An infrastructure breaks down
- An organization (re)configuring some of its systems
- It is decided to refine the architecture model (virtual)

The GG and their rules will be useful to have a precise description of what an SoS of the kind considered should involve and realize, something like this:

- The following goals are relevant to the SoS…
- The SoS shall involve…
- It can be reconfigured as…
- Deployment of… requires…

It can be seen as a way to “standardize” what an SoS of this type is.

In the following we illustrate the initial configuration of a “rescue” SoS. We start with some generic entities from the architecture (organization…) and some capabilities/goals.
2.2.2.3 GG rules to establish the “constitution of rescue SoS”

We first use GG and their rules to model Rescue SoS incremental deployment. The initial status of the architecture consists of generic concepts only.

The initial architecture: generic entities only
(types, classes)

At this point we start with the generic concepts only:
• Generic Goal for the SoS
• Generic Organizations
• Generic Services
• Etc.

The first rule illustrates the delegation of a particular capability/goal to a sub-SoS or a service or system. Deployment with delegation requires adding entities and shifting entities (the goal).
The result follows, in two steps. In the first step we glue onto the initial graph the pre-pattern, consisting of the black and red edges of the rule. Pattern matching is successful, so the rule applies. As a result the blue edges are thus added to the initial graph. The result of this step is shown here, with the colors visible:

The result (2)
In a second step we must remove the part of the rule that specifies removals, namely the red edges. The result follows:

![The result (3)](image)

The next rule we develop illustrates the need to deploy some infrastructure in order to meet the SoS goal.

![Another rule](image)

The rule
The guard is
- \( \text{attrib}(\text{available}) \) and \( \text{attrib}(\text{network}) \) comply
- infrastructure is "working"
Applying this rule to the graph of Figure 20 yields the next step in SoS deployment, namely:

![Diagram showing the result of applying a rule to the graph of Figure 20]

We can also encompass with the same modeling style the following aspects of the SoS:

- Initial Configuration of SoS;
- Deployment of new entity (goal, service, system, infrastructure);
- Deployment of new instances of existing entities;
- Re-configuration to handle alarms, volunteered by the command-and-control center;
- Add or modify goals.

So far we have only seen the “constitution” (Grundgesetz), stating what is legal/illega for this SoS among the possible transformations, for a given SoS architecture configuration.

### 2.2.2.4 Adding guards to control run time architecture evolutions

The GG rules, however, do not tell why and when an enabled rule will be actually activated; to this end, rules are guarded as follows:

- A GG-rule gets *activated* by some *trigger* emitted by some node of the current SoS architecture graph;
- Its application results in some new entities being deployed; the GG-rule also specifies how the attributes of these new entities are *set* or *reset*.

With this additional mechanism of guards, we can capture:

- Initial Configuration of SoS;
- Deployment of new entity (goal, service, system, infrastructure);
- Deployment of new instances of existing entities;
- Re-configuration to handle alarms, volunteered by the command-and-control center
- Add or modify goals.

This mechanism of guards establishes the link between the SoS-wide coordination level and system/service level.

A first rule with guards is shown on Figure 22. It consists in enriching the rule of Figure 19 with guards.

**A transformation rule with guards (1)**

![Diagram showing a transformation rule with guards](image)

Figure 22

GG rules are sleeping by default.

GG rules can be triggered by guards; in Figure 22 the guard is the request of the SoS coordinator to delegate the considered goal to a specific infrastructure. It is thus a trigger from the top-level of the SoS. We may also have triggers by events generated by lower level systems and published at some system interface.

GG rules can cause initialization or reset of new or existing systems.

Our next example of using guards is by adding a guard to the rule of Figure 21 as shown:
The triggering guard is the detection of a fault resulting in a request for repair. As a result of this rule, the communication service shall be restored.

### 2.2.2.5 How to implement GG and GG rules?

When presented as above, GG with their guarded rules are an interesting mathematical model. But we need a concrete syntax for it. We still have to perform a more thorough bibliographical study regarding such existing formalisms. Desirable features for it are the following:

- Guards and init/reset/resume mechanisms can be safely and easily added to a GG formalism;
- GG rewriting rules may be seen too low level, because they are based on enumerated graph configurations;
- Instead we would like to pattern match a set of graphs with 1 rule and get as a result a set of graphs.

Thus we feel we need some kind of high level graph pattern language:

- For tree-shaped architectures a solution exist in the form of Xpath together with Xpath-based tree rewriting rules; key in Xpath is the possibility to refer to descendent nodes, which are star iterates of the child node relation.
- For general graphs, however, no such high-level pattern matching formalism exist; reason for this is the existence of cycles in general graphs.
- A possible pragmatic solution could consist in combining enumerated rules for general graphs together with XPath-based rules for tree-shaped sub-architectures, if any in the considered SoS. This solution is questionable at the moment and remains to be investigated.

This being said, one way of providing a concrete formalism to specify guarded GG-transformation rules is to develop it as a **UPDM view transformation formalism**. Such a formalism could be a specialization of existing **model transformation rules** from MDE environments (as e.g., available from Sodius, IBM). Questions are:

- Is this the right look and feel?
- Do we need to reshape it?
The advantage of this approach is that it sticks to UPDM and preserves layering. An alternative approach could be to rely on more classical syntactic features already in use in Java or any familiar language:

- The "new" mechanism of instantiating a new object from a class

To recover layering we would then need to have the syntax clear enough so we can recover the GG-rule activated from the syntax. If we are unable to do this, then we'll lose the capability of performing architecture-level abstractions for analysis.

2.2.3 Some questions regarding concepts for reuse

2.2.3.1 Services in SoS: anything specific?

XaaS has been a successful buzzword for the last years. XaaS means "X as a Service", where X refers to anything you may want (infrastructure, software, storage, software, communication...). SoA (Service oriented Architecture) is another buzz that is somewhat older. What is the point in distinguishing between services and systems, since both perform some function with some performance characteristics? Thus, we see a benefit in clarifying what we mean by a service in the context of DANSE.

Services are meant to be stateless in that:

- They may possess internal states but they do not expose them on their interface
- Internal states have no effect on call/responses by the service

This allows for many users to share a same service without interfering with each other in an unwanted manner. So, there is no need, for a service, to know who is going to use it. Service interfaces are simpler from functional viewpoint; emphasis is on Quality of Service (QoS).

In contrast, systems can expose to the outside some abstraction of their internal state, for use in interactions with other systems. Therefore, systems know whom they are interacting with. System interfaces or contracts can combine rich functional behaviors with complex QoS aspects; we have studied this in SPEEDS, CESAR, and COMBEST projects.

Services can be orchestrated using activities, see Figure 14. Orchestration languages exist. The industrial standard is BPEL, an OASIS standard (also an IBM product). BPEL offers a number of constructs, including the following essential ones:

- Launching calls in //
- Fork/Merge
- Switch (case statement)
- Sequence

BPEL is the standard but turns out to be a mess. Concepts are not orthogonal and the language has been given a number of different semantics, which is not a good indication.

Orc has opposite characteristics:

- It is a public script language http://orc.csres.utexas.edu/ for writing applications over the Web
- It comes with a nice theory and a clean, usable, semantics.

By being a clean functional language, Orc is nicely embedded in its host functional language Scala and is ready for built-on dynamic creation and reconfiguration. However it is an academic tool and does not support some of the many features found in BPEL.

So, with regard to the question raised in the section title, the answer is probably no – at least assuming that we will be using orchestration languages supporting dynamicity. Maybe this answer is to be mitigated by the
consideration of goals in SoS and how services contribute to them. This latter issue may have something to do with QoS issues in services and orchestrations but all this is open at the moment.

### 2.2.3.2 Systems in SoS: anything specific?

The answer is no, but.

It is no in that we want to reuse systems as they were considered in systems oriented projects such as SPEEDS, COMBEST, and CESAR. Since DANSE is dealing with SoS, we will not consider detailed models of systems such as used in the low-level detailed design phases. We will stick with high-level models of systems and possibly use interfaces and contracts only when dealing with systems. Still, the need for SoS-level simulation may require handling system models beyond interfaces only.

There is, however a but. We have no firm understanding yet regarding what kind of information systems must expose at their interfaces regarding the contribution to the achievement of SoS goals.

### 2.2.3.3 Contracts in SoS: how do they survive?

Extending contract and interface theories of the kind we had in SPEEDS to support dynamicity raises fundamental difficulties. Rather than trying this we propose to stick with what we have so far and apply interfaces and contracts to statically defined entities occurring in SoS, i.e.,

- Systems
- Services
- GG-transformation rules: the rules are meant to be statically defined (unless more is required)

Having these mechanisms, we should be able to

- State contracts on initial architectures;
- State contracts on rules to modify architectures;
- Analyze how “contracts” get propagated through the application of rules.

It may be that we still need some kind of specific architecture-level contract dealing with the top-level. All this remains to be investigated. However the policy we chose, to attach interfaces and contracts to static entities only, seems reasonable.

### 2.2.4 Goals: a proposal

We see goals as properties along with a metric to assess how well these properties are satisfied. Metrics may be multi-dimensional by involving, e.g.:

- Properties expressed in quantitative terms, thus providing built-in metric;
- Qualitative yes/no properties along with a probabilistic assessment of their satisfaction
- Properties expressed in partially ordered domains such as used, e.g., in security
- Cost of achieving a property, how much resources it takes to get them satisfied
- And combinations thereof.
2.2.4.1 What goals should offer as a concept

We begin with a wish list for the concept of goal:

- It should be possible to attach goals to systems, sub-SoS, and the SoS itself. Accordingly, systems enhanced with their goals must be composable.
- It should be possible to have goals quantitative and blend them freely with hard goals of yes/no type; goals can be quantitative in themselves or be made quantitative by assigning to them a probabilistic status.
- Goals should allow for „tradeoffs“ (unlike pure constraints, for which compromise may not always exist).
- Goals can be attached to the architecture model – a GG in our approach.
- Goals should support adaptation & learning in some way.

2.2.4.2 Proposed approach: goals as Gibbs fields on graphs

We propose to borrow widely used techniques from pattern recognition and statistical mechanics, namely Markov random fields on graphs, specified using Potentials, and optimized by means of Gibbs sampling (sometimes called \textit{Metropolis Algorithm}). We stress that this approach is restricted to „static goals“ – it cannot encompass, for example, costs associated with behaviors oft he systems composing the SoS.

Figure 23 shows an SoS architecture. Interpretation of this graph structure is as follows. The SoS comprises three „top-level“ nodes, shown in thick. These top-level nodes can interact directly to concur to the overall goal of the SoS – we define it later. Each top-level node supervises a set of nodes that are not directly visible from the other top-level nodes.

For this approach, we restrict ourselves to SoS goals that can be expressed in terms of a set of parameters or variables attached to each node. We introduce the following notations:

- \textbf{Configuration: } $\omega_x$ denotes an SoS configuration, assigning a value for every parameter in every node $x$ of the graph;
- **Potential**: $V_C(\omega)$ is a potential, assigning, to each pair $(C, \omega)$ consisting of a clique\footnote{A clique is a set of nodes such that any two nodes in the clique have an edge relating them. The graph of Figure 23 contains 4 maximal cliques.} of the graph and a configuration, a real value $V_C(\omega)$ that depends only on the restriction of the configuration $\omega$ to the clique;

- **Energy (as our mathematical notion of „goal“)**: $U(\omega) = \sum CV_C(\omega)$ is the energy of the configuration $\omega$ and constitutes our mathematical instance of a goal in this approach.

**Bibliographical comment**: In statistical mechanics, the following notion of *Gibbs measure* is often considered:

$$P(\omega) = \frac{1}{Z} \exp^{-U(\omega)}$$

where the constant $Z$ (often cased the partition function) ensures that $P$, the *Gibbs distribution*, is a probability distribution over the set $\Omega$ of all SoS configurations. Gibbs distributions are used in statistical physics to study statistical models of ferromagnetism with so-called „spin-glass“ models. We will not be using this latter aspect oft he theory in our development of SoS goals, however.

### 2.2.4.3 Distributed and Autonomic goal optimization

In this section we develop how goals can be optimized in this context. Goal (i.e., energy) optimization is performed, in an autonomic and distributed way using the following algorithm, which is sometimes called „Metropolis“.

For this algorithm, we chose an irreducible symmetric transition probability $Q(\omega, \omega')$, meaning that:

- For every fixed $\omega$, the map $\omega \mapsto Q(\omega, \omega')$ is a probability, which induces a Markov chain on the set $\Omega$ of all SoS configurations.
- $Q(\omega, \omega')$, seen as a matrix, is symmetric;
- Irreducibility means that, from any initial configuration $\omega$, any $\omega'$ is eventually reachable by the Markov chain on $\Omega$.

The optimization proceeds by performing „random iterations“ as follows: a sequence $(\omega_n)$ of configurations is drawn as follows:

- Having $\omega_{n-1}$, we select $w$ at random according to probability distribution $Q(\omega_{n-1}, \omega')$.
- Draw an exponential random variable $Y_n$.
- Set $\omega_n = w$ if $U(w) - U(\omega_{n-1}) \leq T \times Y_n$, $\omega_n = \omega_{n-1}$ otherwise; this means that we do the change if we gain, and we randomly accept the change if we lose. The parameter $T$ that we introduce is called the temperature for reasons related to the reference to statistical physics.

This algorithm is justified by the following theorem:

**Theorem**: the sequence $(\omega_n)$ of configurations is a Markov chain with invariant probability equal to

$$\pi(\omega) = \frac{1}{Z} \exp \frac{U(\omega)}{T}$$
Let us comment on the meaning of this theorem. According to this theorem, the sequence $(\omega_n)$ will spend most of its time in regions where probability concentrates. If temperature $T$ is low, distribution $\pi$ concentrates around the minima of goal $U$.

One important remark must be stated about the architecture of the algorithm. Assume that transition probability $Q(\omega|\omega')$ is local in the following sense: for any given configuration $\omega$, probability $Q(\omega|\omega')$ is concentrated on configurations that differ from $\omega$ within a given single clique only. Thus this transition probability consists in the following two steps:

- Pick a clique in the graph according to uniform distribution over maximal cliques; modifications will occur in this clique only;
- Choose another configuration by selecting, uniformly at random, a different local configuration within this clique.

*When implemented like this, our algorithm takes the form of a multi-agent distributed iteration with an agent assigned to each clique. At each round off he algorithm, a clique is randomly selected by a SoS-level central coordinator, and all other steps are locally handled by agents using local information only.*

### 2.2.4.4 Comments on this approach to handle goals

The above Gibbs framework supports the following:

- Composition of energies formalizes the emergence of an overall goal from local goals
- Quantitative goals, probabilistic goals and hard goals (properties) can be freely blended
- Distributed message passing algorithms allow to optimize the overall goal via local optimization actions performed locally

This framework does *not* support:

- Classical optimal control of dynamical systems from control engineering;
- Goals formulated using behaviors and not only configurations of systems/services attributes.

Thus, our approach for goals suffers from restrictions. Still, we believe that overall, SoS-level goals will not need to involve detailed information collecting behaviors or traces of specific variables. We believe that high-level parameters constituting the SoS configurations will be enough to describe SoS goals and optimize them.

For some cases, SoS-level configurations may have their parameters set by lower level system interfaces. Thus, updating a configuration using our Gibbs algorithm may require a complex action involving the systems involved in the clique selected for the update.

The composition rules for goals and the associated Gibbs algorithms both assume *cooperation* between the entities of the SoS. If a less cooperative approach is wanted, then games may be considered. We are not aware of compositional techniques for games (they may exist).

### 2.2.5 Contribution and positioning with respect to the five dimensions and other remarks

The five dimensions are now recalled:

- Operational independence
- Managerial independence
- Evolutionary development
• Emergent behaviour
• Geographic distribution

We discuss all of them subsequently, with the perspective of the developed approach. Our framework was developed with the following alternative dimensions in forefront, different from the above ones:

1. Architecture management with dynamicity
2. Boundary between SoS management layer and system layer through system interfaces
3. Cross-layer coherent modeling of behaviors, from SoS top-level down to component level
4. Conservative reuse of contracts from systems paradigm
5. SoS-level concept of goal to handle tradeoffs in systems and SoS objectives.

In the sequel refer to each of these dimensions by mentioning their labels from 1 to 5.

2.2.5.1 Operational independence vs. Managerial independence

So far the approach we presented does not by itself ensure full operational independence. Our alternative dimensions 1 and 2 allow for a clean separation of and cooperation between the SoS and system levels. They do not separate the two dimensions listed in the title, however.

Indeed, we do not see any technical need for mathematically separating the two types of independence. For our approach, they are simply referring to different kinds of causes for changing the SoS goals or architecture: by actors of the SoS itself (systems, services…) for the operational independence or by the top-level for the managerial independence.

Syntactic sugar can then be provided for convenience to distinguish the two. We do not recommend a mathematical distinction, however, as this would impair evolutionary development.

2.2.5.2 Evolutionary development

Evolutionary development is very clearly enabled by our approach. Our illustration example shows that the same mechanisms that we offer for dynamic architecture reconfiguration can be also used for SoS construction and configuration. This, we see as a key enabler for evolutionary development. Methodological issues, however, are left aside from our approach. We only proposed a framework. We are not imposing any particular design methodology. But we are confident that our approach complies with the design methodologies considered for DANSE.

2.2.5.3 Emergent behaviour

This is addressed in two ways:

• The ability to perform reproducible simulations is the very first and basic way of allowing the study of emergent behavior as the result of integrating various systems in the SoS. Full semantic consistency throughout all layers supports multi-level simulation.

• The ability to handle trade-offs or conflicts between the goals of different systems or parts of the SoS is the second important aspect of emergent behavior. Our approach and modeling technique for goals with associated multi-agent optimization algorithms allows for this.
2.2.5.4 Geographic distribution

So far we did not investigate yet the mapping of functions or services to resources in a dynamic architecture context. This dimension is thus not addressed yet. We, nevertheless, believe that having a solid mathematical foundation for the SoS framework is a prerequisite.

2.2.5.5 Other remarks

We have proposed a comprehensive approach to handle SoS on top of a firm semantic basis. If achievable, we believe this approach will provide a big step forward in that, for the first time, SoS will be managed in a semantically sound way.

To implement this vision, two major building blocks are needed that may not be available as part of background:

- GG-rules are not provided by UPDM as such. They must be added to UPDM. Now, graph transformations are routinely used in UML and MDE, so it should be cheap developing a user friendly tool implementing GG rules in UPDM.

- Cross-level modeling and simulation is still needed to zoom-in on certain parts of the SoS while keeping the rest abstract. The GG engine does not see inside systems and services. We need this zooming-in with a good semantic quality. The Desyre tool might be able to support this. This would, however, require that Desyre get prepared for general dynamicity, which may not fit the plans of Desyre. As an alternative, we may recommend looking at ReactiveML [http://www.reactiveml.org/](http://www.reactiveml.org/).

2.3 Dynamic Communicating Systems

DCS represent a class of systems which are characterized by their message based interaction. Only assumptions on a (sub-)set of messages associated with a certain states/behavior are made. This is necessary to make any evaluation of interaction protocols. The internal state of such a system represents the role of such systems take during the interaction. That a system has taken a role is determined only by the communication protocol it performs. The individual systems are able but not forced to take a role other than "free agent" which means that this systems do not participate the SoS.

- Emergent behavior:

The assessment of the interaction protocols focuses directly to properties which can not be identified by only taking individual systems into account. Only the interaction (and the associated behavior) for the constituent systems builds the SoS in this context. In the running example a platoon is characterized to have one object with the role "leader" and several objects with the role "follower". The role assignment is based only to the communication protocol. The possibility to verify properties according to the interaction (protocol) leads to a high degree of SoS-ness wrt. this characteristics.

- Geographic distribution:

The car platooning example illustrates the capability of the approach to take any environmental property like detection of a car in front. The idea in this case is that the sensor which detects the other car provides a message containing the id of the car in front. This assumption is necessary to keep the technical and physical issues of sensing and identifying other cars out of the assessment of the communication protocol itself. This does not mean that missing or just erroneous messages could not be taken into account. Also wrt. this characteristics the DCS have a high SoS-ness.
- **Evolutionary development:**

The evolutionary development is not directly address by this approach but the assessment of compromising interaction protocols is -even if not presented in the literature- possible. This allows for example to verify that evolving protocols does not interfere each other. Since this is only possible in a iterative way, the SoS-ness wrt. this characteristics is low.

- **Operational independence:**

The definition of the interaction protocols abstract as far as possible from the details of the operational situation. "Free agents" do not participate on an platoon as long as they do not apply any request according to the protocol. In this example the agents are free to get close enough to participate the platoon. In general these kinds of freedom refers directly to the operational situation. Only for the presented analysis purpose (to verify a certain protocol) the operational independence is limited to the choice of taking a certain role. In general this assumption does not hold but the model supports such kinds of behavior. Therefore the approach itself assumes a high level of operational independence for the individual systems.

- **Managerial independence:**

Since only the ability of joining a SoS is modeled, all other properties of the agents are not taken into account. The platoon example stresses this by only take the ability of sending and receiving messages as well as the application of a certain protocol into account. Any other aspects of the agents are not bound to any degree. This implies that the development and management of this agents is only coordinated wrt. the communication capabilities. This implies a at least medium level of managerial independence.

2.4 Multi-agents

- **Operational independence**

Multi-agent systems’ objectives are really to build decentralized systems based on autonomous behaviors. As a consequence, their design is done in a decentralized philosophy. The reasons for these decentralized objectives are justified by the needs of dynamicity, independence and autonomy intrinsic to such systems. For example, the choice to design a system using a multi-agent approach is usually based on one or several of the following objectives:

- To reduce the complexity of the overall design process, because it is easier to consider the system as a set of autonomous parts. It means that the system is too complex and interdependent to be modeled using a classical top-down approach. Then the designer needs/prefers to mix a bottom-up and top-down approach.
- To build robust systems, because it is easier to consider the system as evolvable and open to make it adaptive to unexpected situations. This time the objective is to design the system as a set of autonomous and interactive sub-systems capable of evolution and reconfiguration.
- To build dynamic/adaptive systems, because it is easier to let the systems decide/learn what are the most efficient situations rather than design statically the entire operational scenario.

Finally, agents are linked to the operational scenario through the goal, role allocations (as described in section 2.2.5), which is a little different from traditional approaches.

- **Managerial independence**

Most of the time, the autonomy and the independence of the agents are managed by designing the interactions between agents. Those interactions are specific to the application to be developed and inspired from cooperative or competitive or any other coordination principles. The definition of these interactions is driven by the knowledge of the designer and by the role and goal allocations. This process is generally strongly iterative, especially when specific emergent properties are targeted in the multi-agent system. In that
case, the system designer chooses a first goal allocation, defines how each agent will achieve those goals and then observes if the required emergent behavior is reached. If it isn’t, then the designer has to re-engineer the role, goal allocations or the way agent are acting to reach their goals or the way the agents are interacting. At this stage, there is no universal engineering solution to trace the goal and role allocations in order to assist this iterative process, but methodologies, such as KAOS, address this point.

- Evolutionary development

The evolution and the dynamicity of multi-agent systems are supported by artificial intelligence techniques. These behaviors are not easy to design nor to integrate, because artificial intelligence learning algorithms are sometimes difficult to tune.

- Emergent behavior

This is often a core property of multi-agent systems, but there are no strong theories that explain how to manage it. Usually, the way it is done is to identify what the role and the goal of each agent are, then to associate their decision and perception models. As described previously, the engineering of the systems is then iterative and progressive.

- Geographic distribution

This property is intrinsic to the multi-agent approach, by definition a multi-agent system has to be decentralized and distributed.
3 Brief conclusion

This document summarizes SoA for SoS and draws various proposals for comprehensive seamless approaches. Mention is done of the requirements on the project as elaborated in WP3. Next step is then to decide which approach to follow, or how to combine them in a richer comprehensive approach.
References


[TWR, 2010] Toben, Tobe and Westphal, Bernd and Rakow, Jan-Hendrik; *Spotlight Abstraction of Agents and Areas; Quantitative and Qualitative Analysis of Network Protocols Dagstuhl Seminar Proceedings*; 2010

[CESAR, 2011] *Specification using Models and Modeling Methods* - deliverable from the CESAR project (European project)

[Respect-IT, 2007] *A KAOS tutorial* - Respect-IT

[DARPA, 2011] DARPA AVM META project summary and deliverables. IBM Section.  