Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS

Guidelines for Architectural Modelling of SoS

Technical Note Number: D21.5a

Version: 1.0

Date: September 2014

Public Document

http://www.compass-research.eu

Contributors:

Stefan Hallerstede, AU
Finn Overgaard Hansen, AU
Claus Ballegård Nielsen, AU
Klaus Kristensen, BO

Editors:

Stefan Hallerstede, AU

Reviewers:

Ken Pierce, UNEW
Lucas Lima, UFPE
Uwe Schulze, BRE

Document History

Ver	Date	Author	Description
0.1	02-07-2014	Stefan Hallerstede	Initial document version
0.2	13-07-2014	Stefan Hallerstede	Reworked Guidelines
0.3	14-07-2014	Stefan Hallerstede	Continued ...
0.4	15-07-2014	Stefan Hallerstede	Finished
0.5	17-07-2014	Claus Ballegaard Nielsen	Added material on systems engineering and SoS modelling
0.6	16-09-2014	Stefan Hallerstede	Reviews comments incorporated
0.7	17-09-2014	Stefan Hallerstede	Reviews comments incorporated
1.0	17-09-2014	Stefan Hallerstede	Final review by PGL incorporated

Abstract

We describe a process that facilitates the analysis of SoS requirements and architectures to be used in early stages of engineering an SoS. The main purpose of the process is to elaborate the description and understanding of the SoS. This provides essential input for subsequent planning and decision making.

Contents

1 Introduction

This document describes a process for high-level architectural SoS analysis. In the preceding deliverable D21.2 [HNH+13] we tried to avoid specifying a process to follow and only gave a set of guidelines. This was motivated by the goal to make the approach widely applicable and, in hindsight, by our limited understanding of the specific problems associated with SoS architectural modelling. The major in-sights that we have gained since then by applying and evaluating the guidelines of D21.2 can be summarised as follows:

- SoS specific problems are non-technical. Most problems occur through interactions of stakeholders, their business policies and legal constraints of various sorts.
- To analyse SoS, architectural outlines are sufficient. Formal modelling can help in evaluating alternatives and their ramifications. Architectural styles are suitable to address sub-problems.
- It is unlikely that a set of capabilities and requirements will be satisfiable as a whole. Instead, depending on different system configurations different sets will be satisfied. An important result of the modelling is to determine these sets and document them.
- Neither CSs nor the SoSs as a whole can be classified to belong to fixed categories. Different aspects of CSs and SoSs can be classified instead. For instance, an SoS can be at the same time virtual, collaborative and hostile.
- It is reasonable to assume that SoS development begins “in the middle”. Some constituents are ready, some are missing and some new capabilities and requirements are to be satisfied. That is, neither a top-down nor a bottom-up approach would be applicable for the development.
- The important aspect of integration to consider is planning of verification and validation. As with most proper engineering tasks related to the actual SoS development, traditional system engineering can solve detailed planning and execution. In fact, this turns out to be the guiding motive of the new guidelines: After an initial phase of SoS modelling and analysis, development activities become system engineering rather than SoS engineering. That is, in later phases engineers do not have to be aware of the SoS. The business and legal constraints of the SoS will have been translated into technical constraints. Long-term maintenance will always involve high-level modelling and analysis being translated to technical problems, where the constraints increase with the lifetime of an SoS in general.

2 Process for Architectural Analysis of SoS

Figure 1 depicts the modelling and analysis process for SoS architecture and integration. The process is iterative. The figure shows the main phases. The concrete architecture of the SoS may vary from the outline architecture produced in this process. The outline architecture is closely related to functional block diagrams customary in system engineering. However, it already specifies architectural styles and communication protocols to be used. These are concepts well-understood by hardware and software engineers. We believe the approach will also be applicable to other engineering areas, but this was not part of this investigation.

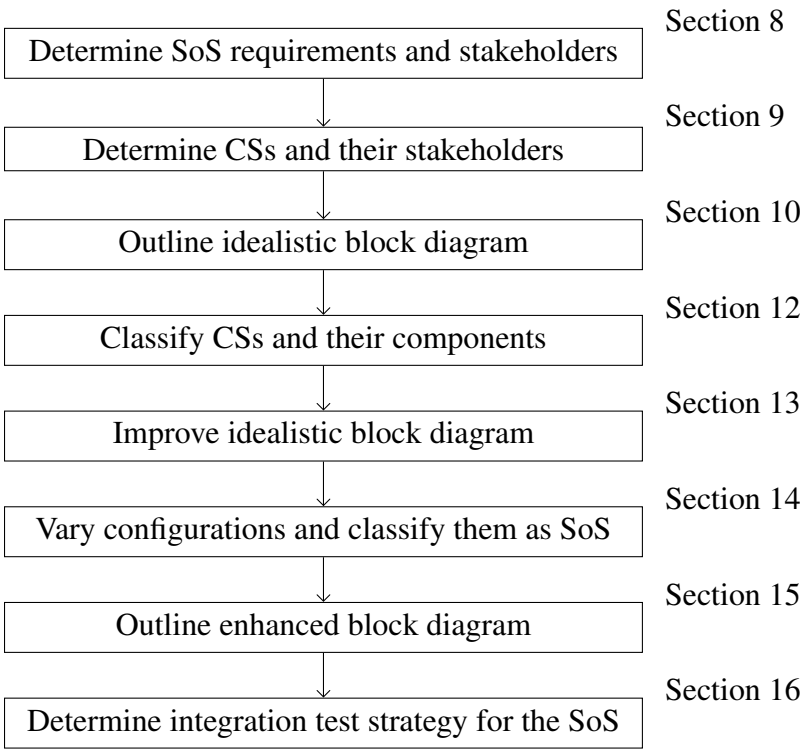

Figure 1: SoS architectural modelling and analysis process

The section with the corresponding descriptions of the process activities are indicated on the right in Figure 1. The remaining sections provide background information and comment on the use of SysML and CML. The sections mentioned in Figure 1 should be read in that order. The remaining sections can be skipped at first reading, but may be helpful for understanding the context.

3 Audience and Aims

These guidelines are addressed to system engineers who have to plan the development of an SoS. As such the guidelines have two aims:

- provide a method for analysing high-level SoS architecture with respect to requirements and capabilities to be satisfied; and
- divide the task of describing an SoS to a set of tasks of describing CSs providing a point early in a project where standard system engineering can be applied thus reducing the complexity of the initial challenge.

The guidelines are not intended to deliver a concrete architecture for an SoS but an outline of an architecture in terms of CSs, architectural styles to address different challenges of the SoS and suggest protocols to interface the CSs. An important outcome of the analysis is a grouping of requirements and capabilities that can be satisfied in different SoS configurations and an associated rating for degraded operation attainable in such configurations.

4 Outcome of the Analysis

The guidelines do not prescribe the concrete shape of the documentation to be produced but focus on the analysis to be carried out. The following six results are expected to be included in the documentation.

1. Outline architecture of the SoS composed of
   - CSs
   - Components
   - Interfaces
   - Protocols
2. CS and component classification
   - Closed
   - Legacy
   - Extensible
   - Open
   - Forbidden
3. Description of SoS configurations and their classification
   - Directed
   - Acknowledged
   - Collaborative
   - Virtual

    - Hostile
4. Model-based high-level requirements validation
5. High-level analysis of SoS properties
    - Evolution
    - Emergence
    - Dynamicity
    - Distribution
    - Openness
6. SoS specific test plan

5 Chief Concerns and Challenges of SoS Modelling

The needs and capabilities that an SoS has to meet are largely dependent on the stakeholders involved in the creation of the SoS. The SoS itself will have certain capabilities that it is expected to deliver, in order to establish an assembly of CS that is valuable enough for the stakeholders of the individual CS to justify their constituent systems' participation. In order to participate in the SoS, a CS may need to make changes to its interfaces or implementation, additionally stakeholders may need to change business principles and to commit resources in relation to both computing power and man power. As such the stakeholders not only have to consider technical challenges, but also to a large degree have financial and commercial aspects, which they need to consider when involving their CS in an SoS.

These considerations are however not the same among the different stakeholders, as they depend on the role of the stakeholder in the overall SoS. This role is dependent on the commitment and interest of the stakeholders in the overall SoS, and on the degree to which the CS needs to be altered in order to be integrated with the other CS in the SoS. The commitment and interest in the SoS will vary as a result of the functionality that the stakeholders' CS deliver to the SoS. They may simply supply a COTS product to the SoS, but besides this they have no further insight or interest in the SoS. However, his does not mean that these stakeholders do not have a lot of impact on the SoS architecture. Such a stakeholder may have a dominant product that they deliver to a large range of other companies, which have to make use of this product due to customer demands. They may also be the keeper of an interface standard, which is so widely used that they have no purpose, interest or even reasonable way of changing it. This gives these stakeholders a large influence on the SoS architecture, although they play a more passive part in the SoS development.

In order for the SoS to be created, a greater interest in establishing the SoS and gaining the increased functionality will be brought by other stakeholders. Their interest and commitment will entail an increasing willingness to change and adapt their own system in order to establish the SoS. An example of this may be a smaller manufacturer that wants to participate in an SoS in order to increase the functionality and value of their own CS. Therefore they will be more willing to adapt and integrate their system in accordance with the interface and communication paradigm of the other CSs.

In engineering projects that involve CSs in which stakeholders are competitors, the SoS development is further challenged as the degree of openness and collaboration between the stakeholders will be limited by commercial and political constraints. In these situations the integration of systems can be hindered by the stakeholders only supplying an interface with very firm restrictions on its use. This creates situations where certain needed functionality may be available on a CS, but may not be used by other stakeholders due to restrictions outlined in the conditions of the interface. The difference in the degree of openness that the individual CS have towards the integration into the SoS is detailed further in Section 12.

These relationships between the stakeholders present the chief concerns and challenges of SoS modelling, as a large degree of the initial challenges in establishing an SoS at the architectural level comes from the business cases, commercial interests and policies of the stakeholders. Dealing with the complexity of having business and legal contraints influence the system design and induce technical constraints are well-known challenges in the field of systems engineering. The existing methods within systems engineering are well suited for dealing with the planning and execution task involved in the development of individual systems. The tasks of SoS Engineering are therefore to 1) identify the operational requirements of the SoS and relevant the CS requirements and constraints, and 2) to obtain the information from the different stakeholders that is needed to outline architectures of possible SoS configurations, which can then be evaluated against the operational requirements.

A structured approach needs to be used to identify and analyse the CS and their stakeholders, as well as for the examination of architecture configurations for the SoS, through modelling. The approach presented in Section 1 and detailed in Section 8 to 16, will enable the development activities to make a transition from SoS development to systems development, which will allow for established System Engineering methodologies to be applied.

6 System Engineering

The successful engineering of systems is a highly complex task in which many constraints have to be considered as a result of requirements, business goals, scheduling, technology trade-offs and costs. In an attempt to deal with the complexity of systems and enable the successful realisation of systems, the field of Systems Engineering (SE) is concerned with establishing and executing interdisciplinary processes to strengthen the design, development and maintenance of systems. SE is focused on choosing the most appropriate methods and tools for concrete systems development projects, as well as on applying these to improve the decision-making process and ensuring the correctness of system specification and design in relation to the constraints placed on the project.

The main organisation for SE, the International Council on Systems Engineering (INCOSE) advocates an engineering approach where systems are at the centre of attention and have a focus on embracing the many diverse fields of engineering involved in a system's life-cycle. They offer tools, processes and mind-sets focusing on the overall system design, from the conceptual design to the systems disposal.

In the INCOSE handbook for Systems Engineering [INC10] the six stages of the Systems Engineering life-cycle is used, as adopted from the ISO/IEC 15288 Standard for systems and software engineering life cycle processes:

Concept Development: Identification of concepts, constraints, trade-offs and operational requirements in order to establish a high level conceptual understanding of what a system is, what it does and how it is supposed to develop over time. The results of this stage are system requirements and an analysis of feasible system architecture solutions.

Development: a viable system architecture is designed from which functional systems can be developed, integrated, tested and validation can be performed.

Production: the manufacturing process of the developed system in which production issues may be identified and the verification of system requirements is performed.

Utilization: transition of the developed system into service and the initiation of the processes developed for system operation.

Support: maintenance and processes to ensure the continued operation.

Retirement: processes related to termination of services and disposal of the sys-

tem at retirement.

The main task of SoS Engineering falls within the Concept Development stage; with the main focus being on the identification of operational requirements of the SoS and the outlining of architectures of possible SoS configurations. At this stage the needs and capabilities of the SoS are analysed and architectural styles and design patterns are used to create "blueprints" of different architectural configurations. Development of a concrete SoS can have different contexts and backgrounds often decided by or governed by several factors as, for example, political reasons, existence of legacy systems, commercial competitions etc. These are all aspects that need to be considered early on in the development cycle.

Both of the deliverable parts D21.5a and D21.5b fall within the Concept Development stage. The guidelines of D21.5a serve to analyse and understand the SoS initially. Usually, the characteristics of an SoS are not known upfront. When they have been clarified D21.5b can be used to define a tailor-made engineering process for the SoS under consideration. This can then drive the development stage.

In the Concept Development stage, SoS engineering is also focused on collecting the interests of stakeholders and in establishing their individual responsibilities. Determining the responsibilities will entail some negotiation between the stakeholders, and consequently a central challenge in SoS engineering is the establishment of collaboration between stakeholders. While a CS in an SoS expresses a large degree of autonomy and are capable of operating independently they will often experience a significant degree of interdependency with the other CS in the SoS. The constituent system will have certain needs, in form of data, functionality, processing power, etc. for which it depends on other CS. Having established as detailed responsibilities between stakeholders as possible, will aid both the integration and operation of the SoS. For some CSs in the SoS the stakeholders may have no interests in the SoS, therefore any responsibilities as well as the integration challenge related to their CS will rely on the collaboration between other stakeholders in the SoS. Again accepting such a responsibility requires negotiation between the stakeholders in the SoS. As such an SoS engineering process should include the possibility of working collaboratively on the system design and analyses.

Finally in this stage, SoS Engineering is focused on specifying and designing test scenarios and test setups. The testing and evaluation of SoS is more challenging than other distributed systems as there is a need for synchronizing test setups, data, results and time across multiple systems and stakeholders, with a lack of centralized management to control the testing process. The testing of the individual CSs can be performed using conventional means of unit, integration and system testing, but the test and evaluation of the overall SoS is dependent on the collab-

oration between stakeholders and their constituent systems in order to coordinate the testing procedure and avoid side-effects from unwanted emergent behaviour. If such a collaboration cannot be achieved between hostile stakeholders this needs to be considered in the test planning.

SoS Engineering will also have a role in the development stage, especially in the integration phase of the stage. For many CSs their development and advancement will however be performed by their individual stakeholders and as such can be handled by SE methods. In the integration phase it will be necessary to perform modifications of CS in order to handle aspects such as data type conversion, communication paradigms and interfaces adaptions. The integration may be challenged by many of the CSs being constrained by design decision made in the past or by the interests of their stakeholders, however such constraints should be identified by SoS modelling and analysis at the Concept Development stage.

No SoS Engineering specific methods are required for handling the technical challenges of the integration in the Development stage, as they are equivalent to those found in other types of systems involving the integration between distributed heterogeneous systems, such as in Enterprise Systems.

7 SoS and CS life cycles

SoS and CSs can be in different stages of their overall life-cycle. The SoSs/CSs can be developed from scratch meaning that there are no pre-existing implementations; they can be mid-life in which systems are operational; and they can be aging meaning that they are borderline obsolescent and require modernisation. Table 1 shows three life cycle states for a given SoS/CS i.e. NEW, MID-LIFE or AGING.

Maturity	SoSE Focus
NEW	Develop a new SoS/CS with no initial capabilities
MIDDLE-LIFE	Enhance an existing SoS capability Develop a new SoS capability Enhance an existing SoS architecture/framework
AGING	SoS currently exists, but needs major restructuring or a new framework/architecture, potentially requiring refactoring of some/all of the CS

Table 1: SoS maturity and SoSE Focus points (Adapted from [LB12])

The vast majority of SoSs can be considered to be in the Mid-life state, even when the SoS is first established. The reason for this is that each CS of the SoS, will be providing some functionality to the SoS, giving the SoS an initial capability. It will be rare for an SoS to be considered NEW as that would entail the SoS being built from scratch to form a system in which none of the CSs provide any functionality for the SoS. Evidence of this is found in the SoS characteristic: independence. For the CS to operate independently from the SoS they must have functionality independent from the overall SoS design.

Figure 2 shows how a given SoS can be characterized to be in one of the three SoS maturity states and for each of these, the actual CSs can as well each be characterized to be in one of the three states NEW CS, MID-LIFE CS or AGING CS .

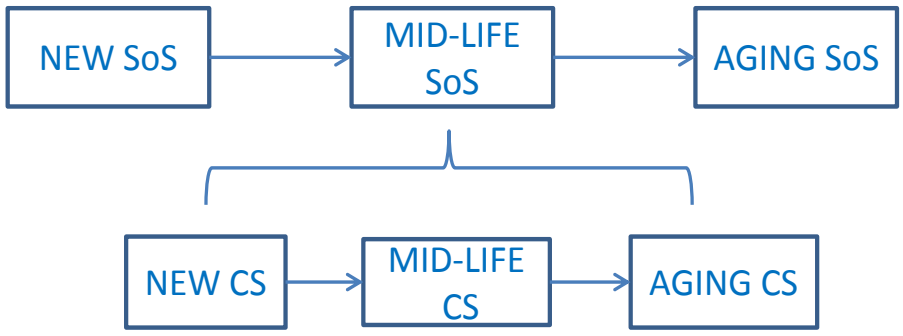

Figure 2: SoS and CS lifecycle maturity states

As the individual CSs composing the SoS may be at different states in their respective life-cycles it brings along the risk of having non-parallel cycles and asynchronous development paths. This makes the capabilities of overall SoS, as well as the evolution of these, unpredictable. Models of the CS can be used to perform simulations which can be used to assess these risks.

The Middle-in development context describes an SoS that is composed of a number of CSs that each deliver a functionality which when combined enable the SoS to meet its goals/operational requirements, as illustrated in Figure 3.

Focusing on the Middle-in system development context, an SoS may evolve further by either: 1) a new CS being added to the SoS, 2) a CS being removed from the SoS or 3) a CS being replaced with a new version.

When adding a new CS to the SoS, the dynamicity of the overall SoS has to be considered: 1) how dynamic does the architecture need to be? 2) what is the frequency of changes? 3) how are new communication paradigms handled?

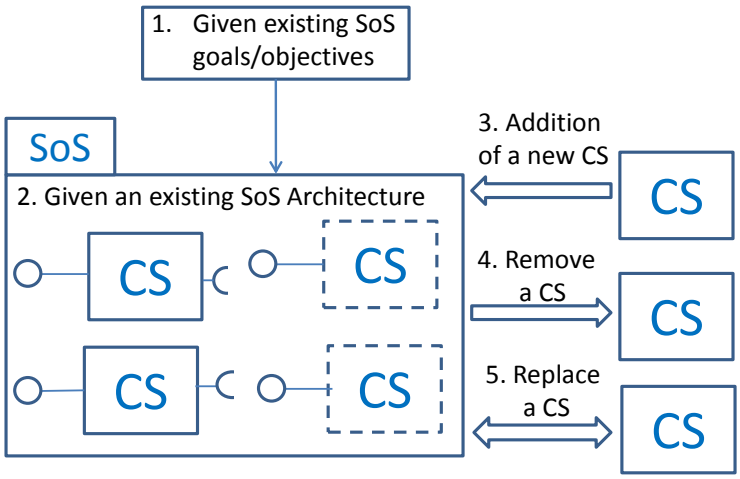

Figure 3: Middle-in SoS Development

and 4) how can the SoS be tested following the addition of unexpected emergent behaviour or system errors?

In the removal of a CS from the SoS, the error-handling, integrity and performance of the SoS needs to be considered: 1) how is the loss of a CS handled, 2) how is data message loss dealt with and is data integrity ensured? if so, how is it ensured, 3) what does the removal of certain CSs mean for the overall SoS ability to meet the operational requirements?

The replacement of a CS with e.g. an improved or alternate version of the CS, can either be performed by the removal and addition of a system, or by swapping the system during run-time while maintaining state. The way replacements of CSs are handled within the system architecture needs to be considered by examining the different possible system architecture configurations. In addition to this, the replacement of an existing CS is an alternation of the SoS for which it can be more difficult to determine the impact for the SoS, as opposed to the addition or removal of an entire CS. Considerations needs to be made on how the effects of replacing a CS can be identified, for instance via testing strategies or the use of modelling and simulation.

As such the middle-in development context includes both the concepts of evolution, and emergent behaviour, which are shortly introduced below:

Evolution The concept of system evolution applies both at the SoS-level and the CS-level. An SoS and its CSs may have long life times, with each CS often in

a different stage of its individual system life cycle. Evolution is natural for these long-lived systems, where changes can result from technological changes, new or changed user capabilities, or new legal requirements, e.g. government legislation.

Emergence The concept of emergent behaviour applies only at the SoS level. It is a characteristic that emerges at the SoS-level as a result of the interaction between a number of CSs and is a behaviour which cannot be achieved by, or attributed to, any of the individual CSs.

An SoS engineer needs to include the considerations presented in this section when performing the needed assessments for the approach presented below.

8 Determine SoS requirements and stakeholders

We assume that the first steps of moving from needs and capabilities to requirements are carried out by methods from SoS requirement engineering such as described in deliverable D21.1 [HPHH12]. This step mostly involves translation of business needs into verifiable requirements. We consider needs to provide a "pull" on the system: requests for the provision of services. Capabilities are considered to provide a "push" of the system: services are provided to gain competitive advantages. Although capabilities may satisfy needs, this is not a necessity. By tracing requirements to needs and capabilities we can identify a preliminary set of stakeholders for each requirement. This is going to be refined by tracing requirements further to CSs that are also associated with stakeholders. At this time we do not take into account the CSs. This analysis can be carried out using SysML Requirement Diagrams in order to keep a record of the reasoning and tracing information.

9 Determine CSs and their stakeholders

The understanding of the SoS is still incomplete. We expect the CSs identified at this stage to grow in the two subsequent steps producing SoS block diagrams. Understanding the CSs, their relationships and stakeholders is one of the major objectives of the process. This has far-reaching consequences on feasible sets of requirements, degraded operation and on SoS integration. Whereas the block

diagrams are produced in SysML, the shape of the list of CSs with attached documentation can be maintained in any suitable form. Although there is a large variation in CSs a few items can be prescribed:

- significance for the SoS;
- maturity;
- evolution to date and foreseeable;
- stakeholders;
- provided features, interfaces and protocols;
- components of the CSs;
- legal and technological constraints; and
- standards and compatibility.

Two principle kinds of stakeholders must be distinguished: technological and business. Many of the constraints seen in CSs are not technological in nature but related to business goals and interests. We collect the terms like societal, legal, i.e., "non-technological" in the one term: *business*. The characterisations of CSs and CSs tend to be driven by business constraints. It is advisable to consider business interests of the stakeholders first and defer technological considerations to a later point. Whereas for technological problems technological solutions are often not far away, business constraints may disallow such solutions.

This activity needs to be carried out continually while the architecture of the SoS is developed and made gradually more specific.

10 Outline idealistic block diagram

The high complexity of a typical SoS makes it difficult to reason about its architecture directly. In order to approach a complete model of the SoS we first make some simplifying assumptions: We assume that all features of all CSs are immediately accessible and all interfaces and protocols are compatible. Essentially, we assume that we are analysing an ordinary system. The SoS constraints are ignored. Using the idealistic block diagram we argue that the architecture outlined by it satisfies the requirements. This provides assurance that the CSs of our initial list are, in principle, sufficient to realise the SoS. In the next steps SoS characteristics are taken into account. In general, this will require changes to the

idealistic block diagram. In particular, additional CSs may be required to manage cooperation between CSs.

If a requirement addresses a CS directly, the CS stakeholders can validate it at the CS level using an appropriate approach. Whether this is the case or not may depend on concrete configurations of the SoS architecture. We do not consider such requirements here as they cannot be properly associated with the SoS.

If a requirement spans several CSs, it always concerns the architecture in the following way: it is necessary to put forward an argument for why the requirement is satisfied by the SoS. This argument will invoke all CSs that are necessary in order to realise the requirement. Such arguments serve two purposes:

- They provide an informal proof of the fitness of the SoS for the required purposes.
- They help in identifying CSs (and stakeholders) that are involved in the realisation of that specific requirement.

Usually the argument will depend on known and specified functionality of the CSs and take account of the interfaces and communication techniques employed. This can be seen as a "test" of these entities. Usually, this will point to problems of the current architecture that need to be solved among the stakeholders.

11 Use of SysML for SoS Descriptions

In this section we illustrate how SysML can be used to describe SoS architectures. Idealistic block diagrams and other diagrams mentioned later can be conveniently represented in SysML. We suggest four concrete diagrams to be used:

- Context Diagrams,
- Use Case Diagrams,
- Block Definition Diagrams,
- Sequence Diagrams.

These diagrams begin as rough sketches and are refined during the modelling and analysis. Eventually, this process will produce an outline architecture of the SoS by way of which requirements satisfaction can be analysed and integration can be planned.

11.1 Context Diagrams

Context diagrams determine the set of actors which are external to the SoS and representing either user roles, hardware units or other systems interacting with the SoS. External systems characterized as actors are not recognized as CSs, as a CS is defined to be a part of the SoS, which means it is inside the SoS context. Figure 4 shows an example of a SysML SoS context diagram.

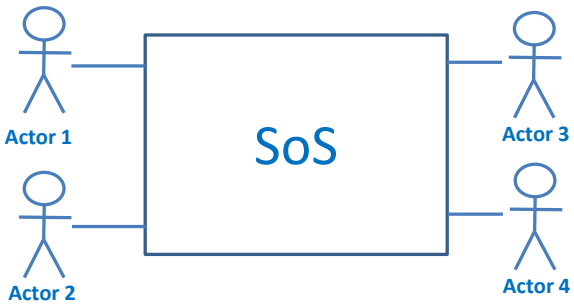

Figure 4: SysML SoS Context Diagram

11.2 Architecture-centric Use Case Diagrams

We assume that the SoS functional requirements are described as Use Cases and optionally documented on a SysML Use Case diagram (see Figure 5). If this assumption is not satisfied it is strongly recommended to develop such Use Cases describing the SoS functionality and optionally show these on a Use Case diagram. Based on these SoS Use Cases and diagrams, a selection of the set of Use Cases that have an impact on the SoS architecture modelling seen from the functional perspective are selected. In particular, Use Cases only concerning a single CS are discarded. They are not considered in the analysis of the SoS as such but should be delegated to the engineering of the corresponding CS.

The result will be a list of architecture-centric Use Cases. These use cases are also the foundation for the simulations in CML. The abstract SoS models expressed in CML should exhibit behaviour satisfying the functional requirements expressed by the Use Cases.

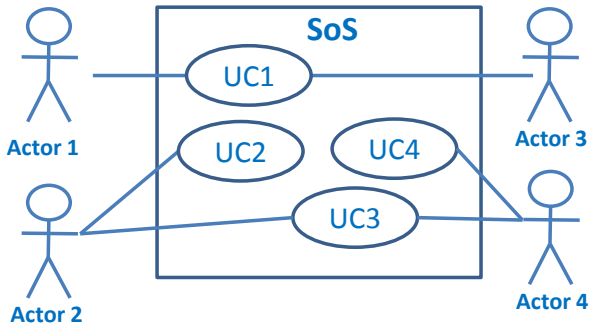

Figure 5: SysML Use Case Diagram

11.3 Block Definition Diagrams

Figure 6 shows an example of a SysML block definition diagram, where an SoS "Use Case 3" (SoS UC3) is allocated to be satisfied by three CSs i.e. CS1, CS2 and CS3.

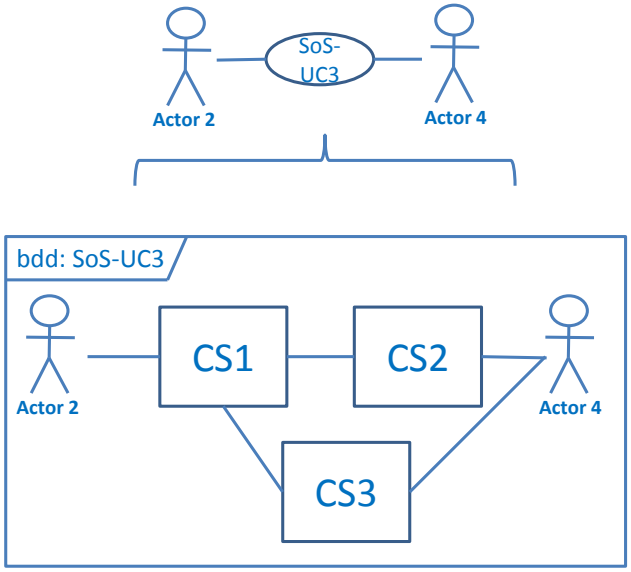

Figure 6: SysML Block Definition Diagram for SoS-UC3

11.4 Sequence Diagrams

The SoS-UC3 behaviour can be described using a SysML sequence diagram as shown on Figure 7, where the interactions are shown between the actors and the

identified CSs. If some of these CSs have state dependent behaviour a SysML state diagram shall be provided.

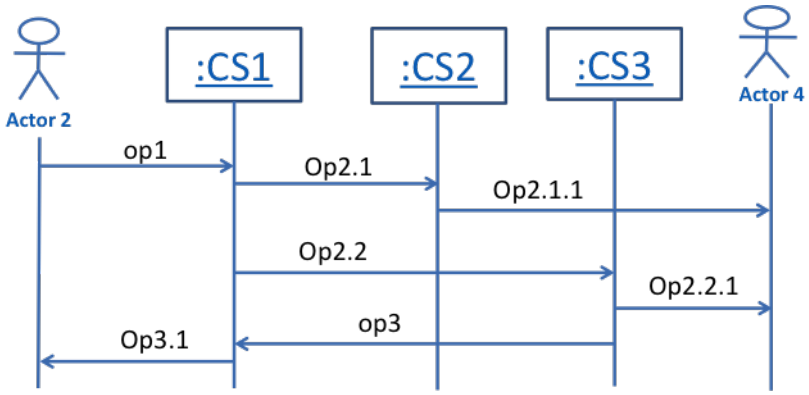

Figure 7: SysML Sequence Diagram for SoS-UC3

11.5 Analysis of SysML SoS Descriptions

SysML descriptions of the SoS are useful to understand and discuss the structure of the SoS. By means of its formal CML semantics it does provide sufficient operational insight for this purpose. SysML/CML is suitable for simulation or forms of static verification such as proof or model checking. Based on Use Cases (or Use Case Diagrams) concrete scenarios can be produced that are subsequently simulated on the formal CML model of an SoS (see [Pet13] for an earlier approach using VDM++). The results of the simulation can be used to produce evidence of requirements that are fulfilled and serve as a basis for discussion with stakeholders. Based on this, needs, capabilities and requirements can be discussed and improved or adapted with respect to what can be achieved. It is not only necessary to explore variations in the architecture of an SoS but also variations in the expectations of what the SoS should achieve. The modelling and analysis may be used to suggest alternatives to stated needs and capabilities.

In particular, simulations can be useful to analyse SoS properties such as those described in Table 2.

12 Classify CSs and their components

The idealistic block diagram shows what is required from each CS in terms of functionality, interfaces and protocols. For each CS it is necessary be determined

Property	Example
Evolution	change of interfaces or their semantics
Emergence (wanted and unwanted)	feature interaction
Dynamicity	dealing with mobile devices
Distribution	conflicting objectives through relevant locations of devices
Openness	extension of protocols or new networking technology

Table 2: SoS properties

how it *could* support all of those, provided the actual functionality, interfaces and protocols *were* compatible. We identify components with different groups of functionality, interfaces and protocols and classify them. We distinguish an SoS into three decomposition layers: SoS, CS, component. The component layer is associated with concrete functionality. The reason for considering components as subdivisions of CSs is that usually each CS has functionality with different degrees of openness. For instance, service discovery may be freely accessible whereas requesting diagnoses of a patient record stored on some device may not be.

A CS and its different components can be classified into the different categories forbidden, closed/legacy, extensible and open. Each category describes the degree of openness towards the integration into a given SoS. The categories are discussed further below.

12.1 Closed

A closed CS or component has to be integrated into an SoS without allowing any changes to be made to the CS. Figure 8 shows a closed CS, which has both a set of provided as well as a set of required interfaces, which must be satisfied by other CSs in the SoS. Commercial examples of closed CSs are the Global Positioning System (GPS) or Google Maps. A CS may be closed but "open to a certain degree" or in the extreme "entirely closed". In practice, many variations exist. We can express "entirely closed" CS as closed CS composed of closed components (see Figure 8). The interfaces defined by a closed component will need to be adapted to by the other CSs in the SoS as the blackbox CS is a type of CS which cannot be changed to suit the SoS, for different reasons. A CS "open to a certain degree" could be closed CS that contains open components. We find that, in practice, back-and-white schemes of characterisation are difficult to apply and lead to a

Figure 8: Closed CS with closed components

choice between artificial decision making and ignoring the guidelines. The only response that can be offered by *practical* guidelines are more flexibility.

12.2 Legacy

Components and CSs that are already long in existence fall into their own category even if, in principle, they could be classified closed, extensible or open. The reason for this is that the development of such components and CSs may long have ended. Legacy components could depend on out-dated technology that makes direct integration impossible and has very limited capabilities as a consequence. Still, it may be necessary to take legacy components and CSs into account. Legacy components and CSs are treated as if they were closed irrespective of their "real" status.

12.3 Extensible

An extensible CS is characterized by allowing (dynamic) installation of components (Figure 9). One possible use of this is to deploy integration code between the extensible CSs and other CSs of the SoS. An example of such an environment is an Android-based CS allowing loading of Android applications onto the CS.

Components themselves may be extensible, for instance, offering a plug-in mechanism. This may add new interfaces or extend existing interfaces. Note that, as in the case of closed CS, the distinction between the CS and components adds some flexibility. For example, an extensible CS may well contain closed components. In general, not everything is extensible in an extensible CS.

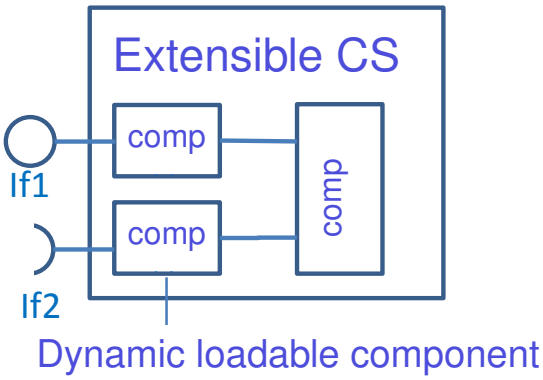

Figure 9: Extensible CS

12.4 Open

An open CS or component permits code changes to enable it to be integrated into a given SoS. This code change may happen at production of the CS. By contrast, extensibility only applies to CSs after they have been produced. Such integrations could, as an example, be performed by the introduction of wrapper components in the CS (see Figure 10).

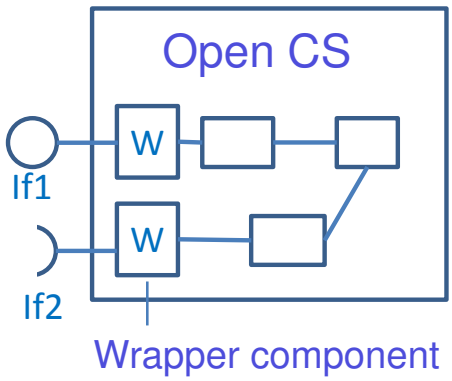

Figure 10: Open CS

Open components may occur as components of the own organisation, as standards or open source software. In particular, for open components and CSs one can see that the category of components and CSs may change in time. E.g., having reached a certain maturity a once open component could become extensible, and eventuallly it will become legacy.

12.5 Forbidden

A forbidden CS or component provides interfaces that can be seen but cannot be used. This will most often be due to commercial or legal reasons. E.g. certain functionality offered by a competitor is protected by IP or some fields of a medical record must remain secret even if they could be read. Forbidden components and CSs are explicitly modelled because during the development stage means must be put in place to ensure that the engineers do not use these. This needs to be managed actively because the corresponding interfaces offered could provide superior functionality to the SoS and could inadvertently enter the SoS. This could render the SoS useless, e.g., because it would break existing laws.

12.6 Summary

Table 3 summarises the different categories and gives a quick overview of their characteristics.

Category	**CS**	**component**
forbidden	access to CS forbidden	component visible but access forbidden
closed/legacy	cannot add, remove or change components	cannot add, remove or change code
extensible	can add components	can add code
open	can add, remove or change components	can add, remove or change code

Table 3: CS and component classification

13 Improve idealistic block diagram

We now have a picture of what the CSs really provide and need to carry that information into the idealistic block diagram. The CS classification determines how this can be done. Depending on how components and CSs are classified: the components can be modified or not; additional components or CSs may have to be included; access to a component may be allowed or disallowed.

For instance, in order to integrate a closed or legacy CS, an additional CS may have to be added so as to not complicate the overall design of other CSs. In order

to integrate an open CS, a component might be added to it or code may have to be contributed to some component. This is the moment when concrete communication paradigms and architectural styles need to be considered. For instance, the additional CS added to integrate a legacy CS could use the blackboard architectural style storing some state of the legacy CS and relaying its functionality. Such a CS may have to bridge communication paradigms or even technological generations. Components can be physical devices, e.g., an SD-card to be "plugged in".

Validation of some requirements may now fail or indicate degraded performance with respect to the stated requirements. We can now explore how different SoS configurations, with certain CSs removed, satisfy different sets of requirements and offer degraded performance of some of these. This is done in the next step.

13.1 Determine Communication Paradigm

Interfaces between the SoS and the CSs depend on the supported communication paradigms. The communication paradigm decides the types of interfaces as well as the communication between the CSs.

Software-oriented SoS offer at least one of the following two communication paradigms:

- A function oriented communication approach as described by e.g. Service Oriented Architectures (SOA) and Web Services (WS), leading to interfaces described by signatures for methods to be called. A function oriented approach is specified using the Interface Definition Pattern, described in [PHP+13].
- A data oriented publish/subscribe communication approach as described by the OMG Data-Distribution Service for Real-Time Systems (DDS) [Gro07, Gro12] leading to interfaces described by a data model based on topics.

More than one paradigm may result due to different decisions made by different stakeholders with respect to different CSs. For the SoS there will be no choice but to integrate all CSs and their communication paradigms.

13.2 Determine Architectural Styles

Once the communication paradigms are known, and the list of architecture-centric Use Cases determined, the SoS can be analysed with respect to suitable architec-

tural styles. As with the communication paradigms some of these may be predetermined by some of the CSs to be integrated.

A set of architectural styles considered appropriate for SoS concerns in general is described in [PHP+13]. Different choices can be made to explore one or more candidates for the SoS Architecture.

The following architectural styles are identified and described in [PHP+13] in relation to SoS modelling. This list is not intended to be complete but gives an indication for the kinds of architectural styles appropriate for SoS modelling and will be a good starting point for exploration of SoS architectures:

- Centralized architecture,
- Service Oriented architecture,
- Publish-Subscribe architecture,
- Pipes and Filters architecture,
- Blackboard architecture.

14 Vary configurations and classify them as SoS

The purpose of this activity is to determine characteristics of the SoS in order to develop integration and verification strategies. An SoS may belong to several categories depending on the different configurations considered. The result of this step depends on the configurations actually analysed. Which configurations should be considered depends on which configurations are possible and what set of requirements they satisfy. Some configurations may be chosen for verification concerns, even if it would never actually occur.

Having the CS descriptions in place allows different SoS configurations to be analysed with respect to the requirements. It must be argued how different requirements are satisfied by different configurations. This can be backed up by providing simulations of high-level SysML and CML models that show conformance or failure to satisfy requirements. The results of this analysis can be fed back to the level of needs and capabilities by the means of tracing information available from needs and capabilities analysis.

In addition to the classification, the result of this task is an analysis of configurations versus requirements providing insight into full performance with respect to requirements, or degraded performance.

This activity is mainly a requirements analysis activity where the SoS requirement specification is analysed with the purpose of supporting the SoS architecture decisions and providing evidence towards the fitness of the SoS for the stated needs and capabilities. Different sets of needs and capabilities should be documented with respect to supporting configurations. This documentation is relevant for the users of the SoS because many of the possible configuration choices will be made during operation of the SoS and not at development or installation time.

14.1 SoS Categories

SoS can be distinguished into the following five prioritised categories with priorities 5 down to 1:

5. Directed
4. Acknowledged
3. Collaborative
2. Virtual
1. Hostile

The first four categories are those suggested by [Mai98] and [DJL08]. We have added the fifth category "Hostile" that is similar to "Collaborative" but assumes that collaboration does *not* happen voluntarily as is often the case when products of competing companies are combined.

Each configuration must belong to one category. If two categories appear possible, the one with the lower number must be taken. For instance, a virtual hostile SoS configuration is hostile. Section 14.7 explains the rationale behind this.

The differences between the categories are used in the enhanced block diagram to indicate suitable architectural styles.

For completeness the definitions of these categories will shortly be repeated here as defined in the COMPASS concept base.

14.2 Directed

An SoS built and managed to fulfil specific goals. Although the constituents can operate independently, within the SoS they accept some central management to ensure that SoS-level goals are met [Mai98].

14.3 Acknowledged

Acknowledged SoS have recognized objectives, a designated manager, and resources for the SoS, however, the CSs retain their independent ownership, objectives, funding, as well as development and sustainment approaches. Changes in the systems are based on collaboration between the SoS and the system [DJL08].

14.4 Collaborative

The SoS has no coercive power over the CS, but they voluntarily choose to collaborate in order to achieve the SoS goals [Mai98].

14.5 Virtual

Virtual SoS lack a central management authority and a centrally agreed-upon SoS-level goal. Large-scale behaviour emerges and may be desirable — but there is no visible active management of the SoS or its goals [Mai98].

14.6 Hostile

The SoS has no coercive power over the CS and they do not voluntarily choose to collaborate in a given SoS to achieve the SoS goals.

14.7 Key Properties of SoS Categories

Table 4 [Pet13] provides an overview of the key properties of SoS pertaining to different categories. The classification of different SoS configurations supports

Category	**Central management**	**Voluntary collaboration**	**Agreed goals**
directed	✓	✓	✓
acknowledged	✓	×	✓
collaborative	×	✓	✓
virtual	×	✓	×
hostile	×	×	✓

Table 4: SoS classification

arguments for satisfaction of requirements and degraded modes of operation. It also provides essential information for the validation and verification of the SoS. Depending on the properties one can rely on, planning of the necessary testing effort and responsibilities can be carried out. We prioritise the key properties in the following order from highest to lowest: 'central management', 'voluntary collaboration', and 'agreed goals'. If a choice between two categories needs to be made the lowest one is chosen making fewest commitments to the SoS.

15 Outline enhanced block diagram

After the preceding analysis steps a block diagram of the complete SoS is produced. This is the completed model of the improved block diagram. The enhanced block diagram must support all configurations. The different configurations should be indicated and their SoS classifications stated.

16 Determine integration test strategy for the SoS

Different configurations have their proper test strategy assigned to them. This permits validation against the requirements even if the full system will not satisfy all of them. The classification of the different configurations indicates feasible strategies for testing. The aim of this activity is to contribute towards test planning focussing on SoS specific issues. This is to be complemented with the customary test planning carried out in system engineering [INC10].

SoS specific issues include:

- Not relying on forbidden components and CSs.
- In hostile configurations testing cannot be expected to be carried out by "hostile stakeholders". The architecture of the SoS needs to reflect this by not relying on "hostile CS capabilities" for essential SoS capabilities.
- Testing needs to consider "hostile functionality" to become unavailable. The users of the SoS must be provided with means to deal with this during operation. This aspect is different from a plain malfunction where a CS or components can be repaired.
- Planning which stakeholder who is responsible for different parts in the test plan if stakeholders can be expected to collaborate.

- Planning testing for evolution, emergence, dynamicity, distribution and openness. In particular, this should take into account the results of the analysis done by means of CML models of the SoS.

References

[DJL08] Judith S. Dahmann, George Rebovich Jr., and Jo Ann Lane. Systems engineering for capabilities. *CrossTalk Journal (The Journal of Defense Software Engineering)*, 21(11):4–9, november 2008.

[Gro07] Object Management Group. Data Distribution Service for Real-Time Systems (Ver. 1.2), 2007.

[Gro12] Object Management Group. DDS website [online]. Available from http://portals.omg.org/dds/, October 2012.

[HNH+13] Jon Holt, Claus Ballegaard Nielsen, Finn Overgaard Hansen, Jim Woodcock, Alvaro Miyazawa, Richard Payne, Juliano Iyoda, and Marcio Cornelio. Initial report on sos architectural models. Technical report, COMPASS Deliverable, D22.1, October 2013.

[HPHH12] Jon Holt, Simon Perry, Finn Overgaard Hansen, and Stefan Hallerstede. Report on guidelines for sos requirements. Technical report, COMPASS Deliverable, D21.1, May 2012.

[INC10] INCOSE. Systems Engineering Handbook. A Guide for System Life Cycle Processes and Activities. Technical report, International Council on Systems Engineering, 7670 Opportunity Rd., Suite 220 San Diego, CA, January 2010.

[LB12] J. A. Lane and T. Bohn. Using SysML Modeling to understand and Evolve Systems of Systems. *Systems Engineering*, pages 1–12, 2012.

[Mai98] Mark W. Maier. Architecting Principles for Systems-of-Systems. *Systems Engineering*, 1(4):267–284, 1998.

[Pet13] Anders Petersen. An evaluation of guidelines for system-of-systems architecture and integration, 2013. Master Thesis. Aarhus University Department of Engineering.

[PHP+13] Simon Perry, Jon Holt, Richard Payne, Claire Ingram, Alvaro Miyazawa, Finn Overgaard Hansen, Luis D Couto, Stefan Hallerstede, Anders Kaels Malmos, Juliano Iyoda, Marcio Cornelio, and Jan Peleska. Report on modelling patterns for sos architectures. Technical report, COMPASS Deliverable, D22.3, February 2013.