AMADEOS SysML Profile for SoS Conceptual Modeling

Paolo Lollini[1(✉)], Marco Mori[1], Arun Babu[2], and Sara Bouchenak[3]

[1] Department of Mathematics and Informatics,
University of Florence, Firenze, Italy
{paolo.lollini,marco.mori}@unifi.it
[2] Resiltech SRL, Pisa, Italy
arun.babu@resiltech.com
[3] Université Grenoble Alpes, Grenoble, France
sara.bouchenak@insa-lyon.fr

1 Introduction

In the European Union FP7-610535-AMADEOS project, a conceptual model for Systems of Systems (SoSs) has been conceived to find a common language allowing experts to collaborate on modelling, engineering, and analyzing SoSs (see public deliverable D2.3 "AMADEOS conceptual model - Revised" [1]).

Analogously to the conceptual model for the architecture of software intensive systems, we separated the description of basics SoS concepts into different perspectives. These perspectives are called viewpoints, each of which is focused on different concerns of the SoS: *structure*, *evolution*, *dynamicity*, *dependability, security*, *time*, *multi-criticality* and *emergence*.

- *Structure*: It represents architectural concerns of an SoS. In particular it defines the manner in which Constituent Systems (CSs) are composed [17] and how do they exchange semantically well-defined messages [10] through their interfaces [22].
- *Evolution and dynamicity*: Dynamicity represents variations to the operation of SoS that have been considered at design-time to reconfigure the SoS in specific situations e.g., either after a fault or after the variation of an external condition [21]. Evolution represents changes that have been introduced later to accommodate modified or new requirements by means of including, removing or modifying system functions [16].
- *Dependability* and *security* [2]: It consists of non-functional critical requirements as availability, reliability, safety, privacy or confidentiality.
- *Time*: It is fundamental since SoSs are sensitive to the progression of time and it is necessary to design responsive SoSs able to achieve reliably time-dependent requirements [9].

This work has been partially supported by the FP7-610535-AMADEOS project.

A. Bondavalli et al. (Eds.): Cyber-Physical Systems of Systems, LNCS 10099, pp. 97–127, 2016.
DOI: 10.1007/978-3-319-47590-5_4

- *Multi-criticality*: It aims at integrating together subsystems providing services with different levels of criticality corresponding to different dependability and security requirements [23].
- *Emergence*: It mainly denotes the appearance of novel phenomena at the SoS level that are not observable at CSs level; managing *emergence* is essential to avoid undesired, possibly unexpected, situations generated from CSs interactions as well as to realize desired emergent phenomena being usually the higher goal of an SoS [14].

In this chapter we will focus on the basic SoS concepts belonging to the different viewpoints and on their semantic relationships, and we will present a SysML profile to represent the conceptual model.

The rest of this chapter is structured as follows: Sect. 2 presents the different concepts defined in a SysML profile to model an SoS. Section 3 describes the structural properties of an SoS in term of architecture, communication and interface. Section 4 defines the concept of evolution related to all changes of an SoS. Section 5 presents the concept of dynamicity that represents the variation to the operation of an SoS considered at design time. Section 6 describes the concepts related dependability, security and multi-criticality aspects. Section 7 describes the global notion of time exploited in an SoS, while Sect. 8 defines the concept of emergence of novel phenomena at the SoS level. Then, Sect. 9 introduces a concrete case study to illustrate the application of basic SoS concepts. Lastly, Sect. 10 provides a brief overview of related works before the conclusion in Sect. 11.

2 Conceptual Modeling Support: The AMADEOS SysML Profile

This section focuses on the definition of a SysML profile as a modeling support for representing the basic concepts for SoS and their relationships. Following the viewpoint-driven approach previously introduced, the concepts and their relationships have been modeled using a SysML semi-formal representation, organized in a profile[1] composed by viewpoint-related packages. To this end, we have defined specific constructs and we have exploited already implemented stereotypes available in other related profiles to support specific viewpoints. Our proposed profile is meant to be used by designers in describing the static SoS structure and its dynamic behavior according to the introduced viewpoints. Such an SoS description can be adopted to be kept consistent across viewpoints by tools and for machine-assisted cross-viewpoint analyses (e.g., finding detrimental emergent SoS behavior).

The SoS profile will be used as an abstract model to represent the topology and the state evolution of an operational SoS. The profile diagrams contain the SoS basic concepts distributed in sub-packages as follows:

[1] https://github.com/AMADEOSConceptualModel/SysMLProfileAndApplication.git - GitHub public link to the AMADEOS SysML profile and the Smart Grid application.

- *SoS Architecture*: describes the basic architectural elements and their semantic relationships.
- *SoS Communication*: provides the fundamental elements in order to describe the behavior of an SoS in terms of sequence of messages exchanged among CSs.
- *SoS Interface*: describes all the points of integration that allow the exchange of information among the connected entities.
- *SoS Dependability*: provides the basic concepts related to SoS dependability.
- *SoS Security*: provides the basic concepts related to SoS security.
- *SoS Evolution*: provides the main elements to describe the process of gradual and progressive change of an SoS.
- *SoS Dynamicity*: provides basic concepts related to SoS dynamicity.
- *SoS Scenario-based reasoning*: provides the basic concepts for supporting the generation, evaluation and management of different scenarios resulting from SoS dynamicity, thus supporting decision-making in an SoS.
- *SoS Time*: provides the fundamental elements to describe time concepts.
- *SoS Multi-Criticality*: provide the basic concepts to describe the multi-criticality aspects of an SoS.
- *SoS Emergence*: provides the main elements to describe the SoS emergence concepts.

It is worth noticing that most of the above packages come from a direct mapping to the views previously defined except for *SoS Architecture, SoS Communication* and *SoS Interface* that all together implement the Structure view, and for *SoS Dynamicity* and *SoS Scenario-based reasoning* that map into the Dynamicity view.

We have implemented the whole profile by exploiting the Eclipse integrated development environment, jointly with Papyrus. Eclipse is an open source environment and offers all the related advantages in terms of cost, customizability, flexibility and interoperability. Papyrus is an Eclipse plugin, which offers a very advanced support to define UML profiles.

In the following sections, we will discuss the key elements of the conceptual model for each identified viewpoint. All the new introduced stereotypes extend the "Block" stereotype of SysML, if not differently specified. For the sake of readability, we will not represent such relations in the SysML diagrams describing the different packages.

3 Structure Viewpoint

The viewpoint of *structure* represents architectural concerns of an SoS. In particular, it defines the manner in which CSs are composed [17] and how do they exchange semantically well-defined messages [10] through their interfaces [22].

The static structure of an SoS is based on the concept of a *Constituent System (CS)*, which is *'An autonomous subsystem of an SoS, consisting of computer systems and possibly of a controlled objects and/or human role players that interact to provide a given service'*. A CS exchanges *information* that is either represented by things/energy or data with its *environment* by means of *interfaces*. The environment of a CS includes

all entities that are able to interact with the CS, including other CSs. In our context, information is a proposition about the state of or an action in the world, which is either an attribute of a physical thing (e.g., temperature of a room) or an attribute of an abstract construct (e.g., execution time of a program).

The interfaces among which the CSs interact one another are the *Relied Upon Interfaces (RUIs)*. As such, the CS *service* – which is its intended behavior – is provided at this interface. RUI is further structured in the *Relied Upon Message Interface (RUMI)* and the *Relied Upon Physical Interface (RUPI)*. RUMI allows for message-based communication of CSs over cyberspace (e.g., the Internet) while RUPI enables the indirect physical exchange of things or energy among CSs over their common environment. It consists of sensors and actuators that take and time-stamp observations of and/or act at a defined deadline on some physical state (e.g., the temperature of a room) in the physical environment according to their design. Environmental dynamics (e.g., heat dissipation through walls) act additionally to other CSs on the physical state. CSs that interact with each other over a common physical environment establish a *stigmergic* channel, i.e., they communicate indirectly over influencing and measuring the physical state. For more details on the interface topic, please refer to [11], and Chapter 2 of this book.

The profile supports the description of the static and dynamic structure of an SoS representing: the basic architectural elements and their semantic relationships; the sequence of messages exchanged among CSs in an SoS; the points of integration, i.e., interfaces, allowing the exchange of information/energy among connected entities.

The structural properties of an SoS are described using three different packages "*SoS Architecture"* (Sect. 3.1), "*SoS Communication"* (Sect. 3.2), and "*SoS Interface"* (Sect. 3.3). The first defines Stereotypes useful to describe the topology of an SoS; the second provides Stereotypes to describe the communication aspects between the Constituent Systems of an SoS; finally, "*SoS Interface"* semi-formalizes internal and external points of interaction of an SoS.

3.1 SoS Architecture Package

Architectural components are defined within the "*SoS Architecture*" package (see Fig. 1). This package extends SysML Block Definition Diagram (BDD) in order to model the topology and the relations of an SoS. Blocks in SysML BDD are the basic structural element used to model the structure of systems (Wolfrom) and they can be used to represent: systems, system components (hardware and software), items, conceptual entities and logical abstractions. A Block is depicted as a rectangle with compartments that contain Block characteristics such as: name, properties, operations and requirements that the Block satisfies. A Block provides a unifying concept to describe the structure of an element or a system: System, Hardware, Software, Data, Procedure, Facility and Person.

This type of diagram helps a system designer to depict the static structure of an SoS in terms of its constituent system and possible relationships.

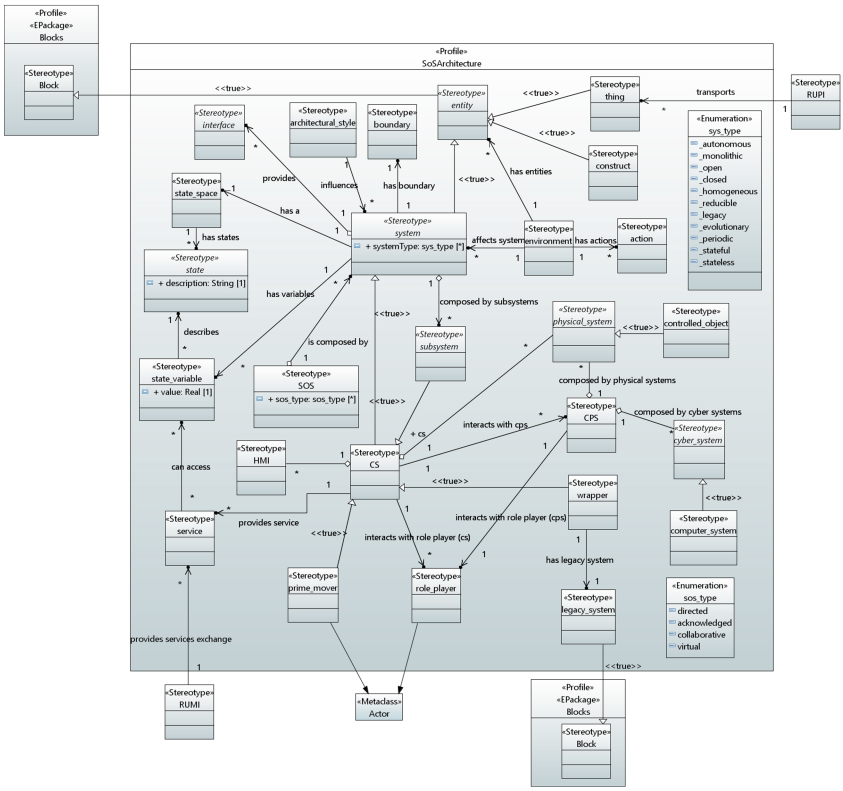

Fig. 1. SoS Architecture package

The first Stereotype is "**entity**" and it extends the SysML metaclass "Block". We distinguish between two different kinds of entities: "**thing**" or "**construct**". They extend the properties of "entity" and so they are also represented as Blocks.

A "**System**" is a type of entity (thereby a Block), it has the same characteristic but it is also capable of interacting with its environment. As it is expressed by the "**sys_type**" Enumeration, a system can be:

- "**autonomous**" - A system that can provide its services without guidance by another system;
- "**monolithic**" - if distinguishable services are not clearly separated in the implementation but are interwoven;
- "**open**" (or "**closed**") - A system that is interacting (or is not interacting) with its environment during the given time interval of interest;
- "**legacy**" - An existing operational system within an organization that provides an indispensable service to the organization;
- "**homogeneous**" - A system where all sub-systems adhere to the same architectural style;
- "**reducible**" - A system where the sum of the parts makes the whole;

- "**evolutionary**" - A system where the interface is dynamic (i.e., the service specification changes during the given time interval of interest);
- "**periodic**" - A system where the temporal behavior is structured into a sequence of periods.
- "**stateful**" (or "**stateless**") - A system that contains (or does not contain) state at a considered level of abstraction.

A system can be influenced by an "**architectural_style**", it can provide a communication "**interface**" and it has a "**boundary**". A "**subsystem**" is a subordinate system that is part of a system and it is related to "**system**" by a composite relation.

A Constituent System or "**CS"** is an autonomous subsystem of an SoS, consisting of human machine interfaces "**HMI**" and possibly of physical "**controlled_object**" and it provides a given "**service**" by interacting with "**role_player**" through the "**RUMI**" (that is introduced in SoS Communication package). RUMI represents a message interface where the services of a CS are offered to other CSs of an SoS, and "**RUPI**" Stereotype represents a physical interface where things are exchanged among the CSs of an SoS. A *wrapper* represents a new system with at least two interfaces, which is introduced between interfaces of the connected component systems to resolve property mismatches among these systems, which will typically be *legacy_systems*. A *prime mover* is a human that interacts with the system according to his/her own goal. In the profile, the "**wrapper**", the "**legacy_system**" and the "**prime_mover**" are "**CS**", which is a Stereotype that extends the property of "**system**" that contains multiple "**sub_system**", which in turn can be "**CS**". A system has a "**state_space**" composed of states described by the variables that may be accessed by the CS service. In addition, a CS interacts with cyber-physical systems. "**SOS**" Stereotype represents the integration of systems, i.e., CSs which are independent and operable, and which are networked together for a period of time to achieve a certain goal. As expressed by the "**sos_type**" Enumeration, an SoS can be:

- "**directed**" - An SoS with a central managed purpose and central ownership of all CSs;
- "**acknowledged**" - Independent ownership of the CSs, but cooperative agreements among the owners to an aligned purpose;
- "**collaborative**" - Voluntary interactions of independent CSs to achieve a goal that is beneficial to the individual CS;
- "**virtual**" - Lack of central purpose and central alignment.

A Cyber-Physical System ("**CPS**") is composed by a set of "**cyber_system**" (i.e., computer systems), and "**physical_system**" (i.e., controlled objects).

3.2 SoS Communication Package

The "*SoS Communication*" package (see Fig. 2) is composed of CSs that exchange information with other elements. In order to represent the exchanged information during the progression of time we use a SysML Sequence Diagram and we represent a CS not only as a Block entity of BDD but also as "Lifeline" metaclass. "Lifeline" is a metaclass

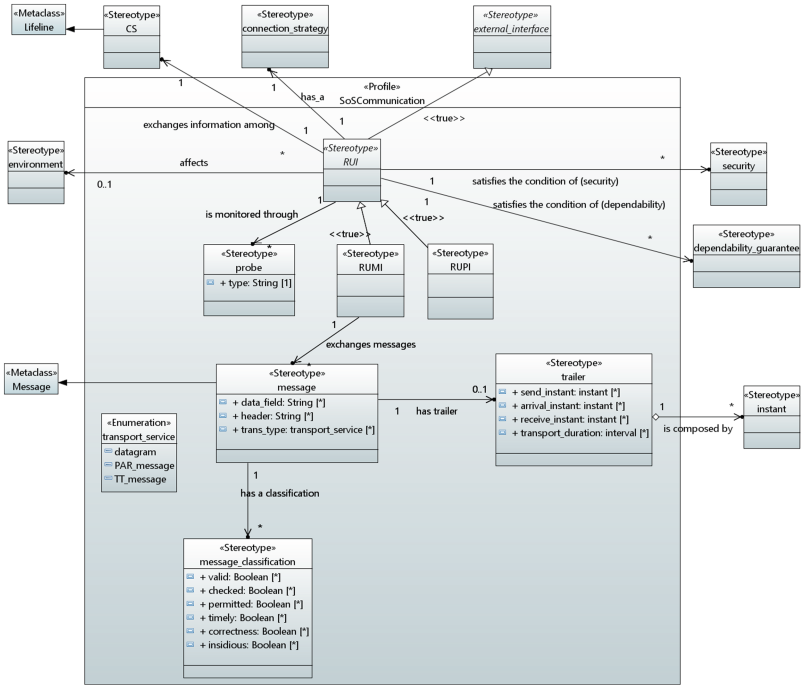

Fig. 2. SoS Communication package

and part of Sequence Diagrams. Through a Sequence Diagram it is possible to represent the behavior of a system in terms of a sequence of messages exchanged between parts and a "Lifeline" defines the individual participants in the interaction (Constituent System). Moreover, through a "Lifeline" it is possible to describe the temporal behavior of an SoS. The time is showed by the length of the "Lifeline" and it passes from top to bottom: the interaction starts near the top of the diagram and ends at the bottom.

A "**RUI**" Stereotype represents an external interface of a CS where the services of a CS are offered to other CSs. It extends "**external_interface**" (defined in "*SoS Interface*" package) and guarantees the exchange of information among CSs ("**CS**" is defined in "*SoS Architecture*" package). A RUI can be represented also as a Sequence Diagram in which CSs are represented by the lifelines that exchange information. A RUI, can be either a "**RUMI**" or a "**RUPI**" and it is monitored through "**probes**". A *RUI connecting strategy* is part of the interface specification that searches for desired, w.r.t. connections available, and compatible RUIs of other CSs and connects them until they either become *undesirable*, *unavailable*, or *incompatible*. A RUI, having a "**connection_strategy**", is instantiated complying to possibly multiple "**dependability_guarantees**" and satisfying "**security**" constraints.

A "**RUMI**" represents a message interface for the exchange of information among two or more CSs and extends the "**RUI**" Stereotype. While messages are exchanged through the RUMI, physical elements are exchanged among the CSs of an SoS through the "**RUPI**"; physical elements are things or energy.

In this package we also model the concept of a stigmergic channel. This type of channel transports information via the change and observation of states in the environment. To represent a stigmergic mechanism, we have introduced the "**environment**" Stereotype that is affected by the RUI.

A message is a data structure that is composed by a "**data_field**", a "**header**" and a "**trailer**" and it flows through a "**transport_service**". The main transport protocol classes to send a message from a sender to a receiver are listed in the "**transport_service**" Enumeration data type, i.e.:

- "**datagram**" - A best effort message transport service for the transmission of sporadic messages from a sender to one or many receivers;
- "**PAR-Message**" - A PAR-Message (Positive Acknowledgment or Retransmission) is an error controlled transport service for the transmission of sporadic messages from a sender to a single receiver;
- "**TT-Message**" - A TT-Message (Time-Triggered) is an error controlled transport service for the transmission of periodic messages from a sender to many receivers.

A message can be classified as:

- "**valid**"- A message is valid if its checksum and contents are in agreement;
- "**checked**" - A message is checked at the source (or, in short, checked) if it passes the output assertion;
- "**permitted**" - A message is permitted with respect to a receiver if it passes the input assertion of that receiver. The input assertion should verify, at least, that the message is valid;
- "**timely**" - A message is timely if it is in agreement with the temporal specification;
- "**correctness**" - A message is correct if it is both timely and value correct. A message is value-correct if it is in agreement with the value specification;
- "**insidious**" - A message is insidious if it is permitted but incorrect.

3.3 SoS Interface Package

The interfaces are the key issue to the integration of systems (see also Chap. 2 of this book) and in this section we introduce an in-depth analysis of the SoS interface concepts, which are represented in Fig. 3.

An interface can be an "**internal_interface**" a "**physical_interface**", a "**message_based_interface**" and an "**external_interface**". The internal interface connects two or more subsystems of a CS (the Stereotype "subsystem", defined in *SoS Architecture* package, is connected with "internal_interface" in order to represent this relation). The physical interface consists of three different types of elements, namely "**sensor**", "**actuator**" and "**transducer**". The "**message_based_interface**" allows the transmission of message by means of "**message**" which are defined in terms of "**message_variable**". Finally, the external interface connects two or more CS (the Stereotype "CS" is connected with "**external_interface**"). A different type of "**external_interface**" is the "**utility_interface**", which is an interface of a CS that is used for the configuration, or the control, or the observation of the behavior of the CS. The

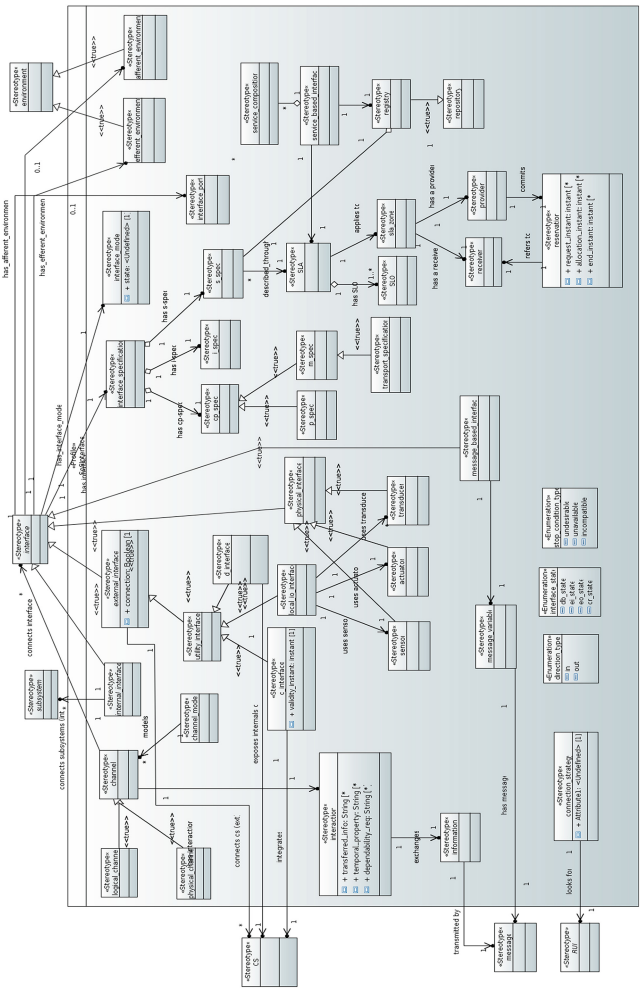

Fig. 3. SoS Interface package

purposes of the utility interfaces are to (i) configure and update a CS, (ii) diagnose a CS, and (iii) let a CS interact with its remaining local physical environment that is unrelated to the operative services of the SoS.

The utility interface is specialized into three different types of interfaces:

- "**c-interface**" - configuration interface - an interface of a CS that is used for the integration of the CS into an SoS and the reconfiguration of the CS's RUIs while integrated in a SoS.
- "**d-interface**" - diagnosis interface - an interface that exposes the internals of a CS for the purpose of diagnosis.
- "**local_IO_Interface**" - an interface that allows a CS to interact with its surrounding physical reality that is not accessible over any other external interface. For example, a CS that controls the temperature of a room usually has at least the following local IO Interfaces: a sensor to measure the temperature, an actuator that regulates the flow of energy to a heater element, and a Human-Machine-Interface (HMI) that allows humans to enter a temperature set point.

An interface has a specification ("**interface_specification**") with different kind of levels: Interface Cyber-Physical Specification ("**cp-spec**"), Interface Itom Specification ("**i-spec**") and Interface Service Specification ("**s-spec**"). "**cp-spec**" is extended by "**m-spec**" that specifies interface properties related to cyber message. "**m-spec**" is further extended by the "**transport_specification**" Stereotype to describe all properties of the communication system for correctly transporting a message from the sender to the receiver(s). "**cp-spec**" is also extended by "**p-spec**" which specifies the interfaces properties related to physical interactions. If the interfaces are service-based, this means that the system provides many services. We have introduced the Stereotype "**SLA**" Service Level Agreement that defines the service relationship between two parties: the "**provider**" and the "**recipient**". "**SLA**" consists of one or more "**SLO**", i.e., the Service Level Objectives. In addition, we have created a new Stereotype that represents the "**reservation**". The reservation is a commitment by a service provider that a resource that has been allocated to a service requester (upon request at "**request_instant**") at the reservation allocation instant ("**allocation_instant**") will remain allocated until the reservation end instant ("**end_instant**"). A "**registry**" contains multiple service specifications allowing multiple "**service_composition**" according to the "**SLA**". A "**channel**" connects interfaces, and it can be physical or logical ("**physical_channel**", "**logical_channel**").

The interaction enabled by the channel has the following attributes: "**transferred_info**" (every interaction involves the transfer of information among participating systems), "**temporal_property**" (an interaction takes time, i.e., for an interaction to occur it is initiated and completed according to system-specific temporal properties) and "**dependability_req**" (e.g., interactions might require resilience with respect to perturbation or need to guarantee security properties like confidentiality). Through channel interactions, the information is transmitted by means of messages". A "**channel_model**" describes the effects of the channel on the transferred information. An "**interface_model**" contains the explanation of the interface. An interface, associated to an "**interface_port**" has an afferent and an efferent "**interface_model**", which

are affected and may affect the interface, respectively. A "**connection_strategy**" Stereotype is defined and connected to a RUI.

4 Evolution Viewpoint

Large scale Systems-of-Systems (SoSs) tend to be designed for a long period of usage (10 years+). Over time, the demands and the constraints put on the system will usually change, as will the environment in which the system is to operate. The AMADEOS project studied the design of systems of systems that are not just robust to dynamicity (short-term change), but to long-term changes as well. *Evolution* represents changes that have been introduced later to accommodate modified or new requirements by means of including, removing or modifying system functions [16].

In contrast to dynamicity, the concept of evolution relates to all changes of an SoS that are not given by design, but arise by changes in the environment (primary evolution), or by new or changed requirements on the SoS service itself (secondary evolution). In the prospect of formalizing a methodology that allows evolution to take place in a controlled manner, the concept of *managed evolution* is most relevant. It is defined as the *'evolution that is guided and supported to achieve a certain goal'* [16];

4.1 SoS Evolution Package

In order to describe this type of processes we have chosen a Block Definition Diagram, because it is designed to show the generic characteristics and structures of a system.

The main SoS concepts are modelled within the "*SoS Evolution*" package of our SoS profile. Figure 4 shows the "**evolution**" Stereotype as a Block of a BDD, aiming at describing an SoS change. In our conceptual model we envision two different types of evolution:

- "**managed_evolution**" - Process of modifying the SoS to keep it relevant in face of an ever-changing environment. Examples of environmental changes include new available technology, new business cases/strategies, new business processes, changing user needs, new legal requirements, compliance rules and safety regulations, changing political issues, new standards, etc.
- "**unmanaged_evolution**" - Ongoing modification of the SoS that occurs as a result of ongoing changes in (some of) its CSs. Examples of such internal changes include changing circumstances, ongoing optimization, etc.

An SoS evolution has a "**goal**", improves the "**business value**" by means of the exploit of "**system_resource**" and can be affected by the environment. Evolution is achieved by modifying CSs and consequently the whole SoS.

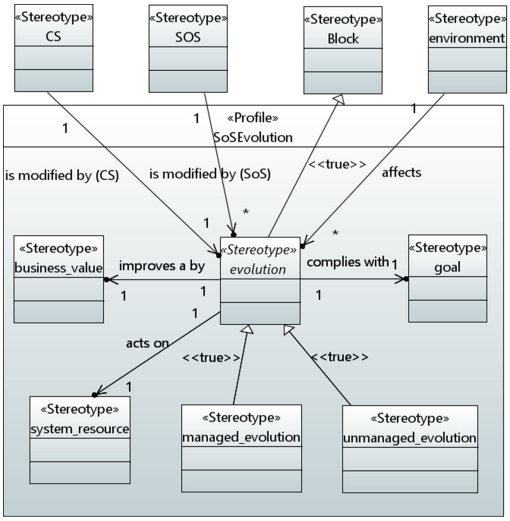

Fig. 4. SoS evolution package

5 Dynamicity Viewpoint

Dynamicity is the property of an entity that constantly changes in term of offered services, built-in structure and interactions with other entities. It represents variations to the operation of SoS that have been considered at design-time to reconfigure the SoS in specific situations e.g., either after a fault or after the variation of an external condition [21]. Dynamicity encompasses all interactions, e.g., message exchange over time.

Closely related to dynamicity is the concept of *reconfigurability*, which is the ability of a system to change its configuration according to the current demands.

The Dynamicity components are described by means of two different packages, i.e., "SoS Dynamicity" (Sect. 5.1) and "SoS Scenario-based reasoning" (Sect. 5.2).

5.1 SoS Dynamicity Package

In this section we show how to use a semi-formal language in order to represent the dynamicity of an SoS. Our objective is to (1) identify which parts of an SoS are dynamic at a certain extent and (2) to represent the dynamic behavior through the interactions among CSs.

As presented in Fig. 5 we have introduced the concept of **"dynamicity"** (belonging to the already defined stereotype **"entity"**), which can be applied either to a CS or to a whole SoS. Dynamicity may be of different nature, either **"dynamic_service"**, or **"reconfigurability"**, i.e., the variation to the CSs architecture, or **"dynamic_interaction"**. Already defined concepts like **"service"** and **"interaction"** are the objects of a dynamic behavior.

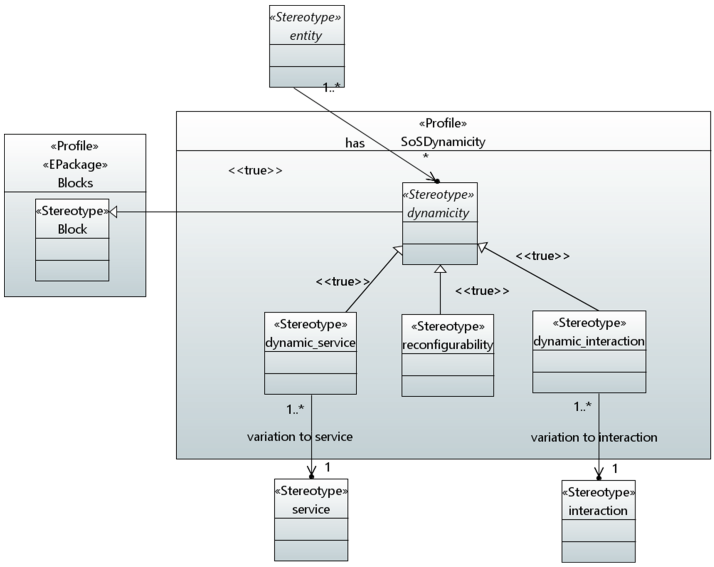

Fig. 5. SoS Dynamicity package

Eliciting dynamicity behavior of different nature that applies to different portions of an SoS is not enough to have a full understanding of the dynamic behavior. With this aim, along with the dynamicity package, we have considered interaction diagrams in order to focus on the message interchange between a number of lifelines: Sequence Diagrams. We propose a methodology to be used to represent dynamicity as it follows:

- Making use of Sequence Diagrams to represent the system behavior in terms of a sequence messages exchanged between parts;
- Selecting the constituent systems involved in the communication;
- Describing the most common interactions.

This type of representation helps a system designer to understand which are the properties of an SoS that are constantly changing and how the SoS can change and rearrange its components. The dynamic introduction, modification or removal of constituent systems can introduce new system behaviors that need to be analyzed.

5.2 SoS Scenario-Based Reasoning Package

Scenario based reasoning package aims at supporting dynamicity and evolution of an SoS. By means of this component of the profile we aim at supporting the generation, evaluation and management of different scenarios thus supporting decision-making in an SoS. As shown in Fig. 6, the main concept of this component is **"scenario"** which is composed by a set of **"scenario_state"** each of which associated to an **"event"** to be applied at each state. A state is in instantiation of a set of **"variables"** which are relevant for the decision-making. Such variables can be extracted by means of an

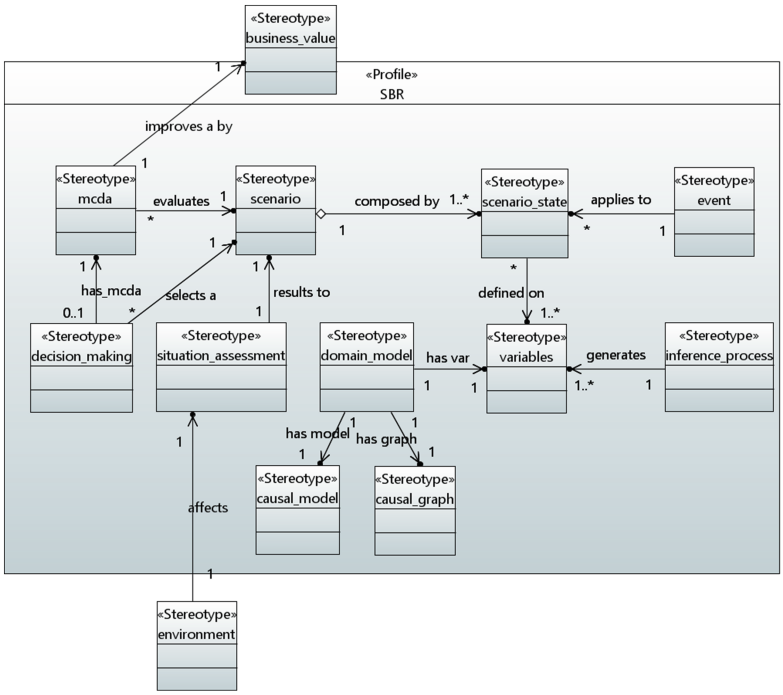

Fig. 6. SoS Scenario-Based Reasoning (SBR) package

"inference_process" and they pertain to a **"domain_model".** The latter defines relationships among variables in terms of correlations (**"causal_model"**) and causation (**"causal_graph"**) dependencies.

The process of generating scenario results from the **"situation assessment"** that depends on the **"environment"**. **"decision_making"** is the process to select a course of actions among different possible alternate scenarios. A multi-criteria decision analysis **"mcda"** may be also applied to improve the decision-making process. Finally, scenarios are subject to pruning and updating operations in order to discard non-correct or un-likely scenarios and to update scenarios dealing with newly available information.

6 Dependability, Security, and Multi-criticality Viewpoints

In any large system, faults and threats are normal and may impact on the availability, reliability, maintainability, safety, data integrity, data privacy, and confidentiality. Traditional dependability and security concepts [2] like fault, error and failure, have been included in the conceptual model. Dependability integrates the attributes of availability, reliability, maintainability, safety, integrity and robustness, and it can be attained by means of fault prevention, fault tolerance, fault removal and fault forecast.

Security is impacted by threats that impose risks exploiting possible SoS vulnerabilities. It is the composition of confidentiality, integrity, and availability; security requires in effect the concurrent existence of availability for authorized actions only, confidentiality, and integrity (with "improper" meaning "unauthorized").

Confidentiality is ensured by means of encryption. Keys are used for encryption/decryption operations, which can be public or private. In an access control system, the security policy is enforced by what is called the reference monitor, which represents the mechanism that implements the access control model. Authorization assigns permissions, which are defined in a security policy. A security policy relies on trusted systems, which encompass hardware, software or human components.

Multi-criticality aims at integrating together subsystems providing services with different levels of criticality corresponding to different dependability and security requirements [23].

A multi-critical SoS is a system containing several components that execute applications with different criticality, such as safety-critical and non-safety-critical. The architecture of safety-critical applications shall be built taking into account that while some part of the system may have strong safety-critical requirements, other parts may be not so critical.

For example, a railway system is a multi-criticality system, given that it consists of components that deliver services at different criticality levels, e.g., a braking service and a heating service. These components usually adhere to different Safety Integrity Levels (SIL) resulting in a system exhibiting different levels of criticality.

In the following we describe the three different packages supporting the definition of dependability (Sect. 6.1), security (Sect. 6.2) and multi-criticality (Sect. 6.3) aspects. The terminology is based on canonical definitions of dependability and security concerns as defined in [2].

6.1 SoS Dependability Package

Figure 7 shows the key concepts captured within the dependability package. A CS or a whole SoS may require possible multiple "**dependability_guarantee**" through the achievement of possible different dependability "**metric**" by means of possible different "**technique**".

A technique is exploited to reduce the occurrence of faults: "**fault_prevention**", "**fault_tolerance**", "**fault_removal**", "**fault_forecast**".

A "**measure**" represents a property expected from a dependable system expressed in terms of a quantitative "**target_value**": "**availability**", "**reliability**", "**maintainability**", "**safety**", "**integrity**", "**robustness**".

The profile supports the definition of "**fault_containment region**", "**error_ containment**" and "**error_containment_region**". The first contains components operating correctly regardless of any arbitrary fault outside the region. These components may have erroneous output actions that are alleviated with the definition of "**error_containment**", which prevents propagation of errors by employing error detection and a mitigation strategy. This leads to the definition of "**error_containment_region**" which contains more "**fault_containment region**" having "**error_containment**". A Fault

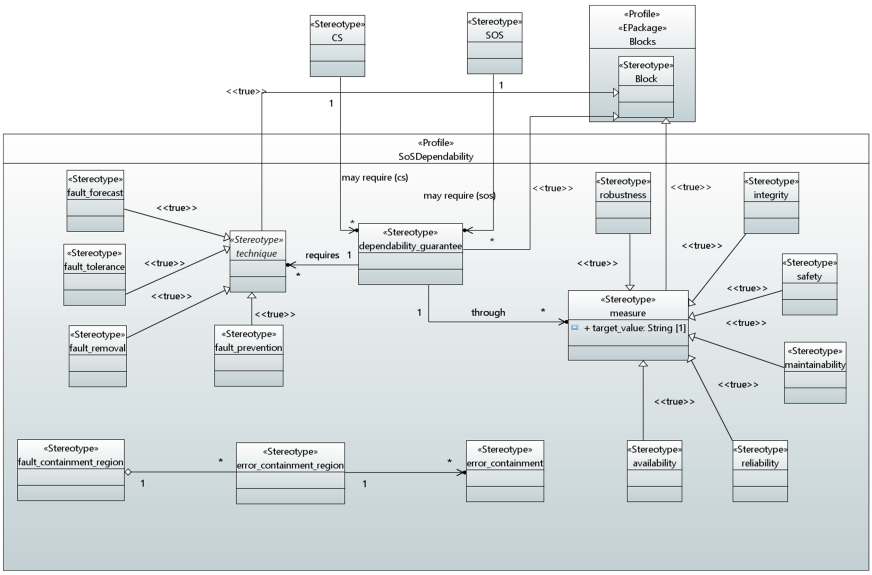

Fig. 7. Dependability conceptual model.

Containment Region (FCR) is a collection of components that operates correctly regardless of any arbitrary fault outside the region.

6.2 SoS Security Package

This section describes the fundamental elements used by a system designer to represent security aspects of an SoS.

As shown in Fig. 8, we connect the Stereotype "**SOS**" and "**CS**" to "**security**" Stereotype to satisfy the security conditions of an SoS. To this end we use "**cryptography**" based on symmetric ("**symmetric_cryptography**") or public key ("**public_key_cryptography**") infrastructure.

The "**encryption**" Stereotype represents the process of disguising data in such a way to hide the information it contains. In this way, data exchanged between Constituent Systems are processed using a cryptography key. Three types of key have been represented: "**symmetric_key**", "**private_key**" or "**public_key**". Symmetric key is exploited for symmetric cryptography, while private and public keys for the public-key cryptography.

The information exchanged (also called "**data**") can be encrypted ("**ciphertext**"), or not encrypted ("**plaintext**"); the "**decryption**" Stereotype represents the process of turning ciphertext to plaintext.

During the cryptography phase the access control ("**access_Control**") consists of a set of actions that are permitted or not allowed by the system. Figure 8 shows a "**subject**" that represents an active user, a process or a device that causes information to flow among objects or changes the system state. A subject may have attributes

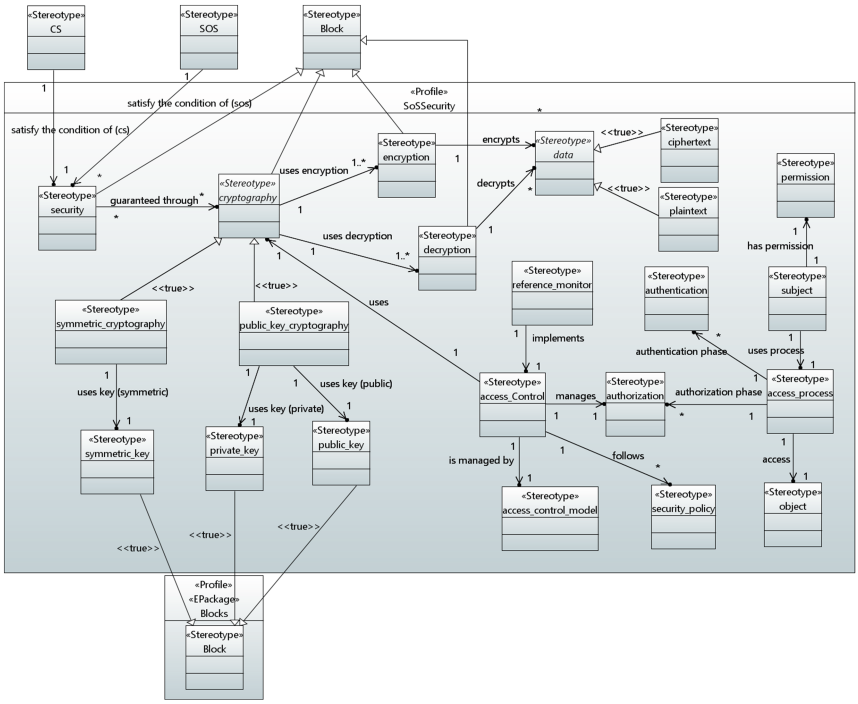

Fig. 8. SoS Security package

("**permission**") that describe how the subject can access to objects. An "**object**" is a passive system-related devices, files, records, tables, processes, programs, or domain containing or receiving information. Access to an object implies access to the information it contains. The "**access_process**" is composed by the "**authentication**" and the "**authorization**". The former represents the process of verifying the identity or other attributes claimed by or assumed of a subject or verifying the source and integrity of data. The latter represents the mechanism of applying access right to a subject.

The "**reference_monitor**" represents the mechanism that implements the access control model and the "**access_control_model**" captures the set of allowed actions within a system as a policy. The access control follows a "**securityPolicy**" that represents a set of rules that are used by the system to determine whether a given subject can be permitted to gain access to a specific object.

6.3 SoS Multi-criticality Package

We introduced the concepts of "**critical_service**" as a particular type of "**service**" having a certain "**critical_level**" (see Fig. 9). The latter is associated to "**dependability_guarantee**" and "**security**". The definition of the stereotype "**service**" belongs to the SoS Architecture package where it is linked to CS, i.e., the component, being

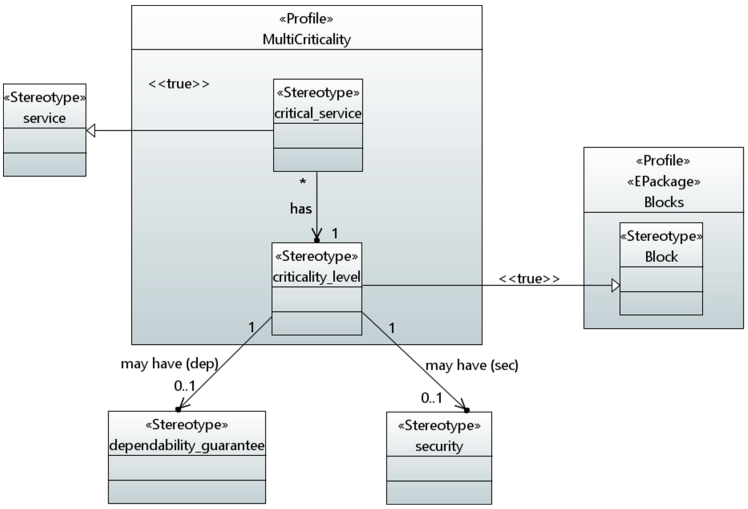

Fig. 9. Multi-criticality package

able to provide the service itself. Thus the concept of "**critical_service**" is indirectly linked to definition of "**SOS**" and "**CS**" by means the definition of "**service**".

7 Time Viewpoint

In an SoS a *global notion of time* is required in order to:

- Enable the interpretation of timestamps in the different CSs;
- Limit the validity of real-time control data;
- Synchronize input and output actions across nodes;
- Provide conflict-free resource allocation;
- Perform prompt error detection;
- Strengthen security protocols.

Time is fundamental since SoSs are sensitive to the progression of time and it is necessary to design responsive SoSs able to achieve reliably time-dependent requirements [9].

The progression of time enables change, i.e., dynamicity and evolution, in SoSs. In the AMADEOS project, it has been concluded that a *global sparse timebase* – accessible by all CSs – is fundamental for reducing cognitive complexity in understanding aspects related to all non-static investigated viewpoints on SoSs. For example, a sparse global time base allows establishing consistently – across all CSs – a temporal order among sparse events, regardless which CSs originally produced these sparse events.

We express the time-related concepts by adopting the MARTE standard [19]. MARTE is an UML profile that provides support for non-functional property

modelling, defines concepts for software, hardware platform modelling, and concepts for quantitative analysis (e.g. schedulability, performance).

We measure time through clocks by defining a clock stereotype that extends the one defined in the MARTE profile. A MARTE Clock Stereotype is considered as a means to access to time, either physical or logical. The MARTE Clock is an abstract class and it refers to a discrete time.

7.1 SoS Time Package

Figure 10 shows a set of main time-related aspects. A Constituent System (defined in *SoS Architecture* package) can share a clock. The Stereotype "**clock**" is also defined as a SysML Block in order to model this concept through a Block Definition Diagram. A (digital) clock is an autonomous system that consists of an oscillator and a register. Whenever the oscillator completes a period, an event is generated that increments the register. A "**timeline**" represents the progression of the time and it is designed with a Stereotype that extends the metaclass "**Lifeline**" of a Sequence Diagram. The "**time-line**" is composed by an infinite number of instants ("**instant**" Stereotype) measured using a "**time_code**" and a "**time_scale**". A time code is a system of digital or analog symbols used in a specified format to convey time information i.e., date, time of day or time interval. A time scale is a family of time codes for a particular timeline that provide an unambiguous time ordering (temporal order of events).

A "**clock**" could be based on an "**internal_sync**", i.e., on a process of mutual synchronization of an ensemble of clocks in order to establish a global time with a bounded precision, or on an "**external_sync**", i.e., on the synchronization of a clock with an external time base such as GPS. It could be a "**reference_clock**", i.e., a hypothetical clock of a granularity smaller than any duration of interest and whose state is in agreement with TAI, or a "**primary_clock**", i.e., a clock whose rate corresponds to the adopted definition of the second (the primary clock achieves its specified accuracy independently of calibration).

Finally, the clock could have the following properties:

- "**accuracy**" - the maximum offset of a given clock from the external time reference during the time interval of interest, measured by the reference clock;
- "**granularity**" - the duration between two successive ticks of a clock; "**tick**" - the event that increments the register of the clock;
- "**offset**" - the offset of two events denotes the duration between two events and the position of the second event with respect to the first event on the timeline;
- "**frequency_offset**" - the frequency difference between a frequency value and the reference frequency value;
- "**stability**" - a measure that denotes the constancy of the oscillator frequency during the given interval of time of interest;
- "**wander**" - long-term phase variations of the significant instants of a timing signal from their ideal position on the time-line;
- "**jitter**" - short-term phase variations of the significant instants of a timing signal from their ideal position on the time-line.

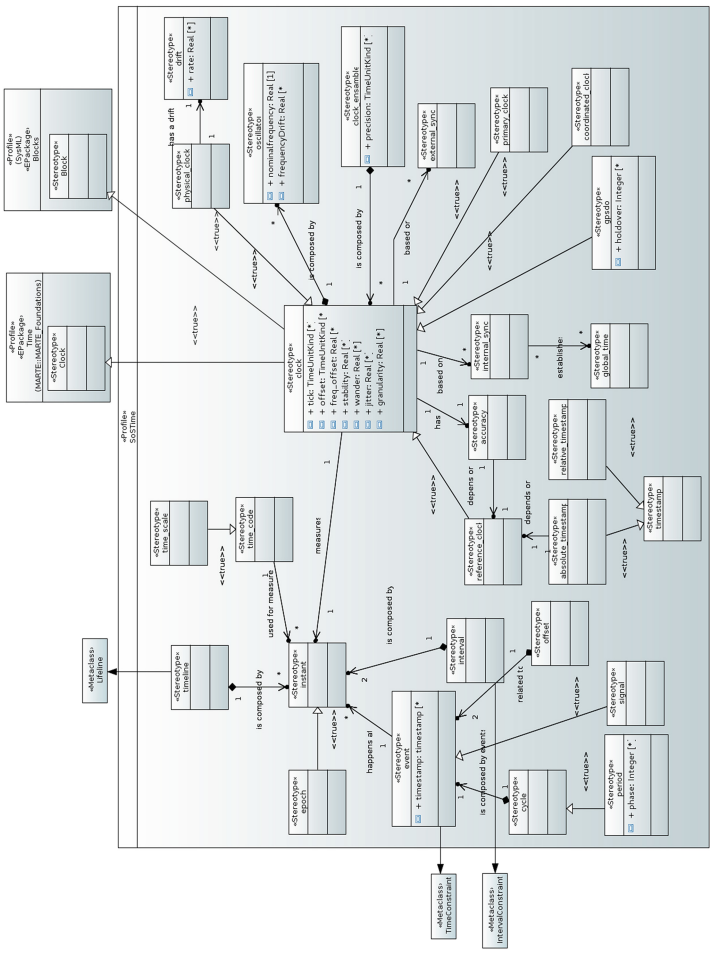

Fig. 10. SoS Time package

If a clock is a "**physical clock**", we use the "**drift**" measure in order to describe the frequency ratio between the physical and the reference clock. A digital clock consists of an "**oscillator**", represented as a Stereotype, with a "**nominal_frequency**" and a "**frequency_drift**", represented as properties. A "**coordinated_clock**" is a particular type of a clock, it is synchronized within stated limits to a reference clock. A "**clock_ensamble**" is a collection of clocks operated together in a coordinated way with a certain "**precision**". We define "**gpsdo**", a Stereotype that represents a particular type of clock where its time signals are synchronized with information received from a GPS receiver, and "**holdover**", a property expressing the duration during which the local clock can maintain the required precision of the time without any input from the GPS.

The "**timestamp**" is the state of a selected clock at the instant of event occurrence. It depends on selected clock and if we use the reference clock for time-stamping, we call the timestamp "**absolute_timestamp**". An ensemble of clocks could synchronize in order to establish a "**global_time**" with a bounded precision.

An "**instant**" is a cut of the "**timeline**" and an "**interval**" is a section of timeline composed by two instants. The latter is defined as an "**IntervalConstraint**" of a Sequence Diagram.

An "**event**" can happen at a particular instant, and to represent this type of information we have used a "**TimeConstraint**" of a Sequence Diagram. A "**signal**" is a particular event used to convey information typically by arrangement between the parties concerned. An "**epoch**" is a particular instant on the timeline chosen as the origin for the time-measurement. A "**cycle**" is a temporal sequence of significant events whereas a "**period**" is a specific type of cycle marked by a constant duration between the related states at the start and the end at the end of the cycle, called "**phase**". The offset of two events denotes the duration between two events and it is represented by the "**offset**" Stereotype.

8 Emergence Viewpoint

The concept of Emergence (see also Chap. 3 of this book) is one of the most important challenges of AMADEOS. As already described in previous sections, SoSs are built to realize new services that CSs separately cannot provide.

Emergence mainly denotes the appearance of novel phenomena at the SoS level that are not observable at CSs level; managing *emergence* is essential to avoid un-desired, possibly unexpected situations generated from CSs interactions and to realize desired emergent phenomena being usually the higher goal of an SoS [14].

In the AMADEOS conceptual model, emergence is defined as follows: '*A phenomenon of a whole at the macro-level is emergent if and only if it is new with respect to the non-relational phenomena of any of its proper parts at the micro level*'. Consequently, it is behavior observable at the global level (e.g., a traffic jam) that cannot be reduced to the behavior of one of the parts (e.g., a single car analyzed in isolation). If an emergent phenomenon can be explained by a trans-ordinal law, i.e., a law that explains the emergent phenomenon at the macro level from properties or interactions of parts at the micro level, it is *explained emergence*. In case such laws have not been found (yet),

it is *unexplained emergence*. While there are cases of unexplained emergence (e.g., the human consciousness), the type of emergence that is occurring in the cyber part of an SoS is *explained emergence,* even if we are surprised and cannot explain the occurrence of an unexpected emergent phenomenon at the moment of its first encounter. If we have made proper provisions to observe and document all interactions (messages) among the CSs in the domains of time and value, we can replay and analyze the scenario after the fact. At the end, we will find the mechanisms that explain the occurrence of the emergent phenomenon. There is no ontological novelty in the interactions of the CSs in the cyber parts of an SoS.

Hence an explained emergent phenomenon can be classified as expected (trans-ordinal laws are known), or unexpected (trans-ordinal laws are not known). Orthogonally, emergent phenomenon can be *beneficial*, or *detrimental*.

Hence four cases of emergent behavior must be distinguished in an SoS. *Expected and beneficial* emergent behavior is the normal case. *Unexpected and beneficial* emergent behavior is a positive surprise. *Expected detrimental* emergent behavior can be avoided by adhering to proper design rules. The problematic case is *unexpected detrimental emergent behaviour*. For an in-depth discussion about emergence in SoSs we refer to [12].

8.1 SoS Emergence Package

In this section we show how to use a semi-formal language to represent an emergent behavior of an SoS. Nevertheless, because of the nature of the emergence concept, defining a semi-formal language, thus only eliciting an emergent behavior, is not sufficient. Our aim is also capturing operational aspects related to emergence by considering an SoS in action.

For these reasons we propose two different types of representation that a system designer can choose:

- Block Definition Diagram;
- Sequence Diagram.

Figure 11 shows the profile package for the emergence behavior as a block definition diagram.

This package represents the main concepts of emergence using a Block Definition Diagram. We represent a "**phenomenon**" as a block and we distinguish an "**emergent_phenomenon**" from a "**resultant_phenomenon**". An emergent phenomenon can be explained ("**explained_emergence_phenomenon**") or unexplained ("**unexplained_emergence_phenomenon**") and in the former case there is a trans-ordinal law ("**transOrdinal_law**") that explains the behavior.

An SoS with emergent phenomena has an emergent behavior that could be expected, unexpected, beneficial or detrimental. For this reason, we consequently defined the four following blocks: "**unexpected_and_detrimental**", "**expected_and_detrimental**", "**unexpected_and_beneficial**", "**expected_and_beneficial**".

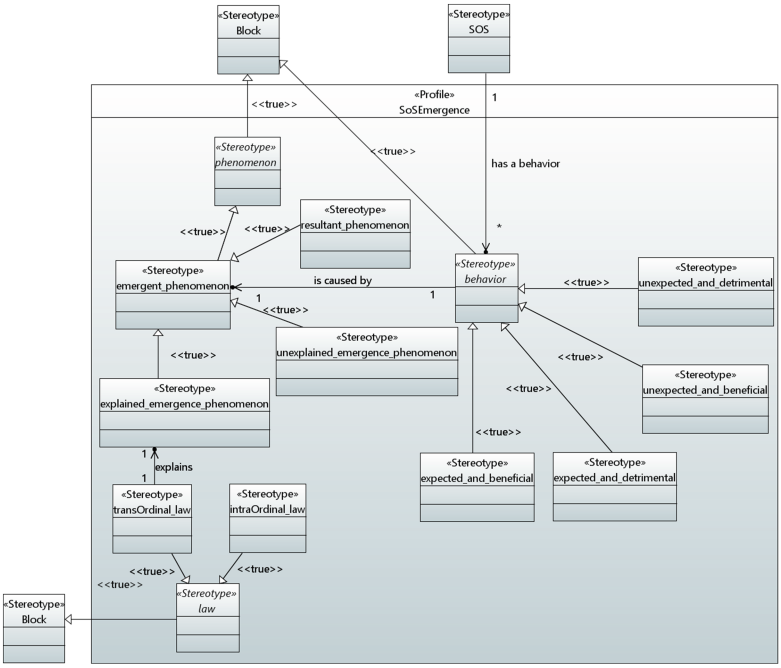

Fig. 11. SoS Emergence package

9 The Profile at Work

In this section we introduce a Smart Grid household scenario to exemplify the application of the profile and to instantiate the basic SoS concepts to a concrete case-study from the Smart Grid domain, focusing on the Architecture (Sect. 9.1) and Emergence (Sect. 9.2) viewpoints. Further examples of application of the profile to the selected use-case can be found in [15] and public deliverable D2.3 "AMADEOS conceptual model - Revised" [1].

In a Smart Grid household scenario different operationally independent subsystems aim at delivering the desired emergent phenomenon of improving the efficiency and the reliability of the production and distribution of electricity through communication facilities. Requests for energy coming from *electronic appliances* are forwarded towards the subsystems in charge of granting or denying each request while achieving the Smart Grid goal, i.e., keeping the production and consumption rates for connected households balanced.

Figure 12 shows the topology of the main subsystems involved within a single household of the Smart Grid scenario. Washing machines and microwaves are examples of electronic appliances. They represent a *flexible load* which may initiate an energy request. The *smart meter* measures energy consumption and production rates; the *Distributed Energy Resource* (*DER*) manages the energy produced through energy generating and storage systems, like wind-powered electrical generators or batteries.

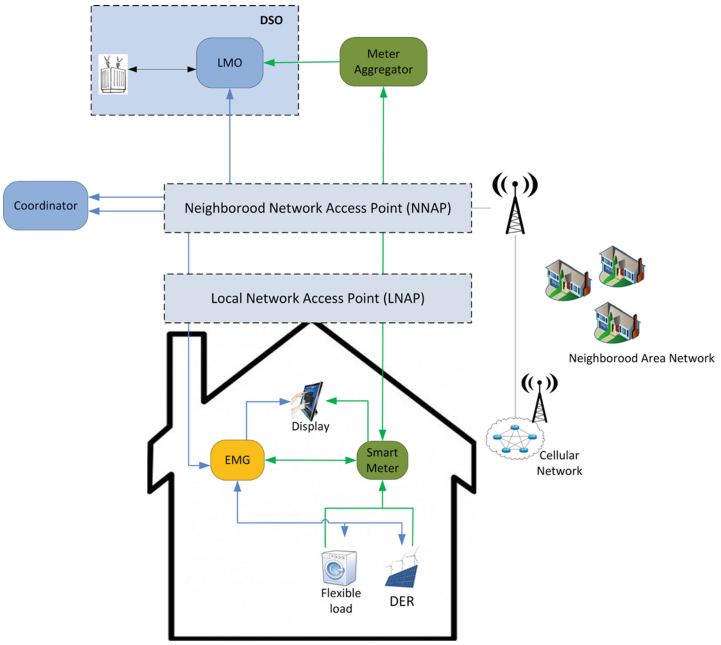

Fig. 12. Smart Grid case study

A *command display* shows consumption rates and enables residents to interact with their own energy control system. The *Energy Management Gateway* (*EMG*) controls the flexible loads and the DER based on measurements received from the smart meter and in agreement with the *coordinator* to establish optimal energy distribution. The coordinator is connected to the *Neighborhood Network Access Point* (*NNAP*) with the aim of keeping the production and the consumption of energy for a set of connected households balanced. A *Distribution System Operator* (*DSO*) regulates consumption and production rates at the country level. By means of its *Load Management Optimizer* (*LMO*), a DSO receives information from a *meter aggregator* and enacts control decisions in cooperation with the coordinator. The access to the household is provided by one or more *Local Network Access Points* (*LNAPs*) connected to a NNAP. All the above mentioned components require proper interfaces in order to exchange control messages and physical energy entities within and outside the household Smart Grid.

9.1 Modeling the Architecture Viewpoint

Using the SoS Architecture package it is possible to represent the topology of any System of Systems. Now we show how to use SoS through the Smart Grid household case study.

First of all, it is necessary to decide what are the main constituent systems involved, and how to represent them. For each system component we use a Block element of a Block Definition Diagram and through the connections we show the relations between

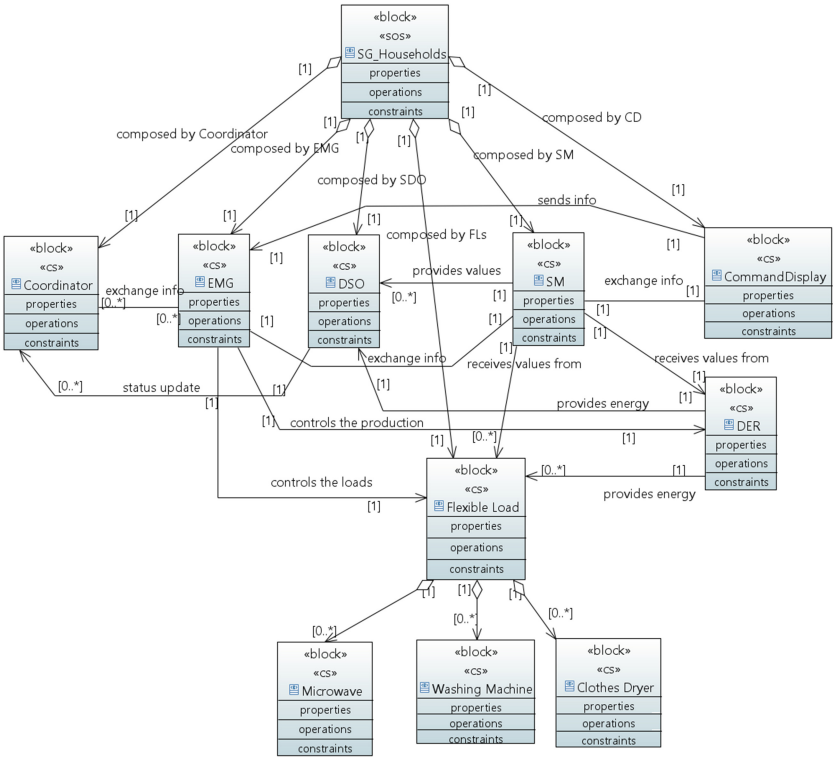

Fig. 13. Smart Grid household Block Definition Diagram

them. Using the stereotypes defined in the "*SoS Architecture*" package it is possible define the Smart Grid household as a system of systems (SoS) and all the other elements as constituent systems (CSs).

Figure 13 shows a model example of a Smart Grid with the application of our profile ("*SoS Architecture*" package). The "SG_Households" is a Block and it is stereotyped as an SoS; it is composed by 5 CSs, which exchange information. Among others, the block "Flexible Load" is stereotyped as a CS and it is composed by a set of household electrical appliances: Microwave, Washing Machine, Clothes Dryer, etc. These latter are switched on and off dynamically based on the current needs.

An application example of the main SoS communication concepts is shown in Fig. 14. Through the Smart Grid household case study, we describe a set of communication messages exchanged between the involved CSs.

First of all, it is necessary to decide which are the involved elements in the communication and how many message are exchanged. We identify a "Lifeline" as a constituent system and a "message" as exchange data between two constituent systems. A message could contain all the properties defined in Fig. 2 and they can be displayed using a constraint or a comment box. Figure 14 shows the message properties using a

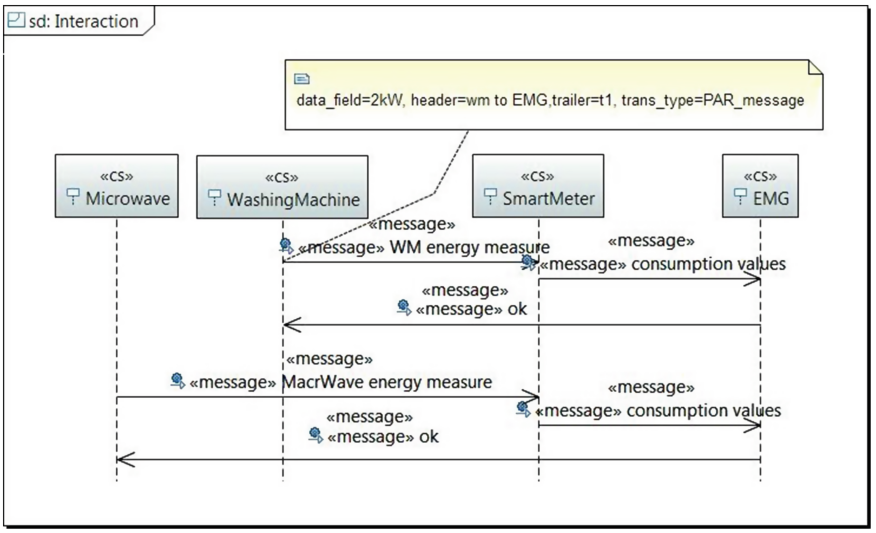

Fig. 14. Message exchange between CSs of a Smart Grid

comment box (e.g., "data_field" = 2 kW, "header" = wm to EMG, "trailer" = t1, "trans_type" = PAR_message).

9.2 Modeling the Emergence Viewpoint

While a Block definition diagram defines stereotypes and related elements to capture statically the emergence behavior, a sequence diagram is able to define dynamic interactions leading to emergence. We adopt a sequence diagram where each lifeline represents a constituent system and each message specifies the kind of communication between the lifelines, the sender and the receiver. An SoS is prone to changes: sometimes constituent systems are incremented, modified or removed. To this end, this kind of diagram helps the system designer to easily update and analyze new system behaviors. The diagram not only describes the communication but it also helps to represent the SoS behavior during the progression of time.

To show the difference between a "static" and "dynamic" representation of emergence, we consider a particular scenario of the Smart Grid previously described. The dynamicity of the household electrical appliances may lead to an emergent behavior of the system in case of a peak of request of energy coming from the neighborhood. Let us assume that because of a public event, an exceptional lighting of specific public spaces has to be supported by the Smart Grid. In this case, while in the household it was commonly possible to turn on microwave and washing machine together, we end up in a very limited provision of energy, which is not sufficient for both the electrical appliances. This phenomenon represents an *emergent behavior* of the Smart Grid since it is not possible to devise it if we only look at the interactions of the internal household CSs without considering the neighborhood CSs.

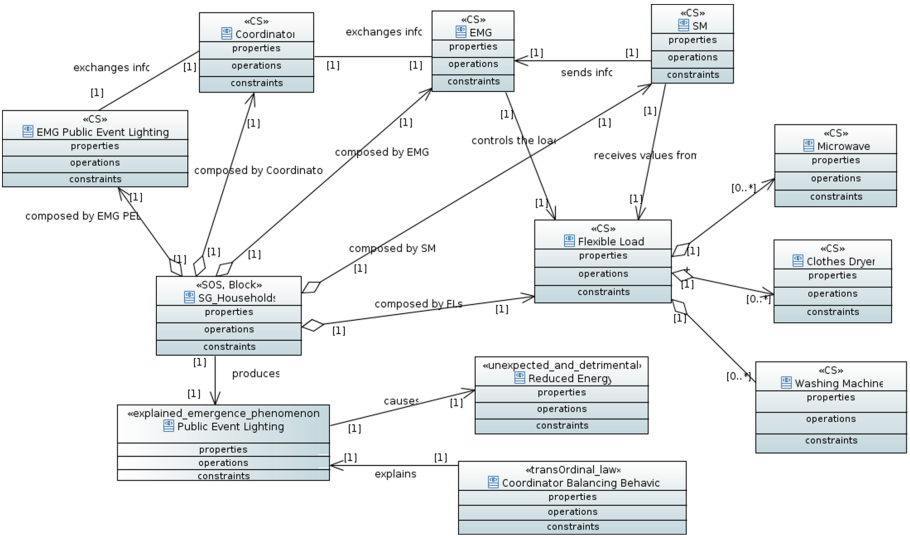

Fig. 15. Smart Grid household – Emergent behavior

Figure 15 shows, through a BDD, how the public event lighting is represented as an explained and detrimental emergent phenomenon, explained by the balancing behavior of the Coordinator and causing reduced energy for the electrical appliances. This phenomenon causes an *unexpected* and *detrimental* behavior of the SoS, which allows it to only satisfy a subset of energy requests.

However, using this type of diagram we are not able to represent the progression of time and the semantic of message that may contribute to reveal emergence phenomena. Especially, the above representation does not attach greater importance to capture the time aspects of the SoS emergent phenomena.

We now introduce the representation of the exceeding peak energy request by using a sequence diagram. We adopt a sequence diagram to show the emergent behavior of the electrical appliances request by means of the interaction among related CSs of the Smart Grid.

As shown in Fig. 16, "electronic appliances" CSs are represented as "Lifeline" and their interactions are represented through directed labeled arrows. Washing Machine is switched on at t2 after the agreement allowed from the Coordinator. As next, the Coordinator receives (at time t3) and grants (at time t4) the energy for switching on the public lighting for the exceptional event. This request is forwarded to the Coordinator by the Public Event Lighting (PEL) EMG, which is external to the household. At time t5, microwave issues its request to be connected to the Smart Grid but it receives a negative acknowledgment at time t7. Usually, the household would be able to switch on the washing machine and the microwave at the same time. On the contrary, because of the public event lighting resulting in a peak of energy from the house neighborhood it results that only a reduced amount of energy is available for the electrical appliance (emergent behavior). Indeed, right before the requests issued from the microwave (time t5) and the clothes dryer (time t8), the Coordinator allocates the energy for the public

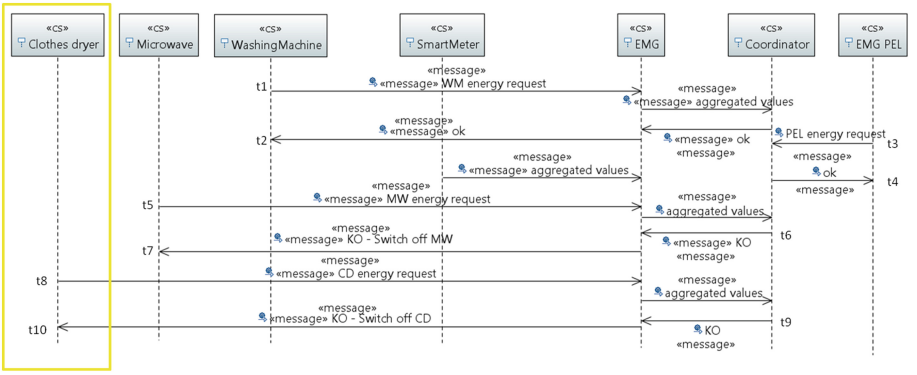

Fig. 16. Smart Grid household SysML model – Emergent behavior description

lighting event and consequently no further requests of energy from the house can be granted.

This illustrative example shows that networked individual systems together to realize a higher goal, which none of individual system can achieve in isolation, could lead to an emergent behavior: impossibility of satisfying commonly granted energy requests. The Emergent behavior is shown through the message exchange and it consists of unexpected and detrimental emergent behavior caused by a system dynamicity property.

10 Related Works

In this section we present an overview of related ADL design approaches presented in the literature of SoSs. This analysis is not meant to be exhaustive but it is based on some of the most representative related works on designing SoSs. Its objective is to determine to what extent viewpoints-based SoS concepts have been already captured in the literature.

In [7] the authors propose the use of SysML in representing an SoS by adopting and in some cases extending canonical SysML diagrams in order to model different viewpoints of an SoS. Beyond *structure*, a specific support to the *multi-criticality* viewpoint is also provided by adopting the specific stereotypes aiming at grouping requirements according to qualitative and quantities metrics to support trade-off analysis. Nevertheless, there is no specific support for other viewpoints, including *time*, *dependability/security*, *dynamicity*, *evolution* and *emergence*.

A partial answer to the above issues is given by the approach presented in [13] providing support to *structure* and e*volution* viewpoints of an SoS by exploiting several SysML models. The authors propose the adoption of diagrams to determine an evolving SoS and its environment and the interactions occurring between an SoS and the environment and among CSs themselves. Noteworthy the approach is still missing specific support to *dynamicity*, *emergence*, *multi-criticality*, *dependability/security* and

time. In [20], the presented SysML modeling approach allows the definition of the SoS *structure* and how to support *dynamicity* and *evolution* viewpoints by means of understanding the dis-alignment of a simulated SoS with respect to its requirements. Noteworthy, it is still missing a specific support to *emergence*, *multi-criticality*, *dependability*, *security* and *time*.

The approaches presented in [3, 8] provide support to model the *structure* of an SoS and *emergence* by means of the extension to SysML diagrams. Analyses of the former models are conducted to provide evidence that requirements are fulfilled. The approach supports fault-handling (*dependability* viewpoint) and responsiveness (*time* viewpoint) of an SoS, but it does not provide any specific support to *dynamicity*, *evolution* and *multi-criticality*.

The approach in [6], within the context of the DANSE EU project [4], supports the definition of an SoS *structure*, *dynamicity* and *evolution* (by means of Graph Grammars), *emergence,* etc., with the only exception of *multi-criticality.* DANSE presented a set of methodologies and tools to model and to analyze SoSs based on the Unified Profile for DoDAF and MoDAF (UPDM). In particular, DANSE focuses on the six models that can be represented as executable forms of SysML as partially reported in [6], according to a well-defined formalism to relate basics SoS concepts and their relationships. In the context of DANSE, the Goal Specification Contract Language (GSCL) assures the achievement of dependability and security requirements and it guarantees the timely response of an SoS.

All these approaches have shown the utility of adopting SysML formalisms to model architectural aspects of SoSs, thus supporting different types of analysis and a first step towards executable artifacts which can be automatically derived. Although these approaches provide detailed insights for different viewpoints aspects, it is still missing (i) an homogeneous synthesis at a more abstract level of key design-related SoS concepts, and (ii) a viewpoint-based vision. Bringing this perspective in one single consistent reference model, it is possible to provide solutions to specific design problems while still keeping the required interconnections among viewpoints.

11 Conclusions

This chapter presented a viewpoint-driven approach to design SoSs by adopting a SysML profile. We pointed out the gaps in the literature of ADLs for SoSs with respect to a set of viewpoints that we deemed essential for understanding SoSs. We outlined the conceptual model at the basis of the profile and we presented how to solve specific viewpoint needs in an integrated fashion by exploiting the high-level SoS representation in a small scale scenario. We implemented the profile in the Eclipse integrated development environment jointly with Papyrus [5], i.e., an Eclipse plug-in supporting advanced facilities for manipulating UML artifacts and SysML profiling.

The AMADEOS SoS profile can be adopted along with a Model-Driven Engineering (MDE) approach. MDE is an approach to system development and it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification. For a model-driven architecture

perspective [18], our SoS profile is a Platform Independent Model (PIM) or, in other words, a view of the system from the platform independent viewpoint. It provides a set of technical concepts involving SoS architecture and behavior without losing the platform independent characteristics.

This kind of independent architecture makes possible to analyze step by step all the PIM viewpoints and to obtain one or more Platform Specific Models (PSM), where the SoS profile is specialized and improved according to the domain/enterprise specific technologies that belong to the enterprise implementing the SoS instance. A PSM is a view of a system from the platform-specific viewpoint. It combines the specifications in the PIM with details that specify how that system uses a particular type of platform and on the platform itself.

Furthermore, the SoS PSM can represent the base step for other activities such as the following:

- Source code generation: through an automatic transformation the SoS model can be translated in source code;
- System analysis: the SoS model can be the starting point for a lot of system analysis like: hazard analysis (HA), Failure Mode and Effect Analysis (FMEA), Fault Tree Analysis (FTA);
- System testing: the SoS model can be the basic layer to identify test procedures or resolve problems of testing coverage.

References

1. AMADEOS project - Public Deliverables. AMADEOS project. http://amadeos-project.eu/documents/public-deliverables/. Accessed 28 Sep 2016
2. Avizienis, A., Laprie, J., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE TDSC **1**(1), 11–33 (2003)
3. Bryans, J., Fitzgerald, J.S., Payne, R., Kristensen, K.: Maintaining emergence in systems of systems integration: a contractual approach using SysML. In: INCOSE (2014)
4. DANSE: DANSE Methodology V2 - D_4.3 (s.d.). https://www.danse-ip.eu
5. ECLIPSE - MDT/Papyrus: Eclipse Model Development Tools (MDT) (n.d.). http://wiki.eclipse.org/MDT/Papyrus-Proposal. Retrieved 14–29 Sep 2016
6. Gezgin, T., Etzien, C., Henkler, S., Rettberg, A.: Towards a rigorous modeling for-malism for systems of systems. In: ISORCW, pp. 204–211. IEEE (2012)
7. Huynh, T.V., Osmundson, J.S.: An integrated systems engineering methodology for analyzing systems of systems architectures. In: Asia-Pacific Systems Engineering Conference, Singapore (2007)
8. Ingram, C., Fitzgerald, J., Holt, J., Plat, N.: Integrating an upgraded constituent system in a system of systems: a SysML case study. In: INCOSE International Symposium (2015)
9. Kopetz, H.: Real-time Systems: Design Principles for Distributed Embedded Applications. Springer, New York (2011)
10. Kopetz, H.: Conceptual model for the information transfer in systems of systems. In: ISORC, pp. 17–24. IEEE Press (2014)
11. Kopetz, H., Fromel, B.: Direct versus stigmergic information flow in systems-of-systems. In: System of Systems Engineering Conference (SoSE). IEEE (2015)

12. Kopetz, H., Höftberger, O., Frömel, B., Brancati, F., Bondavalli, A.: Towards an understanding of emergence in systems-of-systems, pp. 214–219. IEEE
13. Lane, J.A., Bohn, T.: Using SysML modeling to understand and evolve systems of systems. Syst. Eng. **16**(1), 87–98 (2013)
14. Mogul, J.: Emergent (Mis)behavior vs. complex software systems. In: EuroSys, pp. 293–304. ACM (2006)
15. Mori, M., Ceccarelli, A., Lollini, P., Bondavalli, A., Frömel, B.: A holistic viewpoint-based SysML profile to design systems-of-systems. In: International Symposium on High Assurance Systems Engineering (HASE 2016), pp. 276–283. IEEE, Orlando (2016)
16. Murer, S., Bonati, B., Furrer, F.J.: Managed Evolution: A Strategy for Very Large Information Systems. Springer, New York (2010)
17. Nakagawa, E.Y., Gonçalves, M., Guessi, M., Oliveira, L., Oquendo, F.: The state of the art and future perspectives in Systems-of-Systems software architectures. In: SESoS, pp. 13–20 (2013)
18. OMG: MDA Guide Revision 2.0, 18 June 2014. http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf. Accessed 14 Sep 2016
19. OMG: A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2. Document Number: ptc/2008-06-0 (s.d.)
20. Rao, M., Ramakrishnan, S., Dagli, C.: Modeling and simulation of net centric system of systems using systems modeling language and colored petri-nets: a demonstration using the global earth observation system of systems. Syst. Eng. **11**(3) (s.d.)
21. Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., Yan, H.: Discovering architectures from running systems. IEEE TSE **32**(7), 454–466 (2006)
22. Selberg, S.A., Austin, M.A.: Toward an evolutionary system-of-systems architecture. In: INCOSE, pp. 1065–1078 (2008)
23. Verissimo, P.: Travelling through wormholes: a new look at distributed systems models. SIGACT News **37**(1), 66–81 (2006)