AMADEOS Framework and Supporting Tools

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

[1] Resiltech SRL, Pisa, Italy
arun.babu@resiltech.com
[2] Thales Nederland B.V., Hengelo, The Netherlands
sorin.iacob@nl.thalesgroup.com
[3] Department of Mathematics and Informatics,
University of Florence, Florence, Italy
{paolo.lollini,marco.mori}@unifi.it

1 Introduction

This chapter defines the overall tool-supported "AMADEOS architectural framework", with its main building blocks and interfaces. It particularly focuses on Structure, Dependability, Security, Emergence, and Multi-criticality viewpoints of an SoS. Finally, for SoS modeling, a "supporting facility tool" based on Blockly is demonstrated. Blockly is a visual DSL and has been adopted to ease the design of SoS by means of simpler and intuitive user interface; thus requiring minimal technology expertise and support for the SoS designer.

2 Architecture Framework for SoS

Architectural framework does not refer to the specific design of specific system architecture, but they rather represent a view on how such architecture should be described. Although architectural frameworks are "prescriptive" and not "descriptive", there is still no consensus on providing a methodological step-by-step instruction to be followed. In [1], the authors describe a study involving the use of a design approach to guide the development of an SoS architecture by means of rules, guidance and artefacts for collaboratively developing, presenting and communicating architectures without an order set of phases to carry out. In [2, 3], it is noticed a close relation between architecting methods and the architectural frameworks, thus a step-by-step set of instructions is provided to guide the development of SoS architectures.

When building an SoS architectural framework, the aim is to be instrumental in the creation of future evolvable systems of systems. Both description views and methodology shall be allowed, as long as it facilitates the design of the architecture of such systems.

Architectural frameworks that are currently used in SoS literature have been applied in different contexts of operation along with ADL solutions to model different

architectural aspects of an SoS. In the following we provide insights on currently adopted architectural frameworks and ADL approaches.

2.1   ADLs in SoS Architectural Frameworks

This section collects a few ADL approaches that have been proposed in the literature to model different aspects of SoS. They range from approaches dealing with very specific problems to frameworks.

Among the approaches presented in the context of research projects we consider solutions proposed in COMPASS and DANSE EU projects. COMPASS aims at supporting the application of formal analysis tools and techniques at the level of the graphical notations used in current industrial practice. COMPASS project exploits the Artisan Studio tool [4] in order to support system and requirements modelling using SysML as well as software modelling using UML and code generation. As stated in [5] COMPASS proposes the adoption of *Context Diagrams*, *Use Case Diagrams*, *Block Definition Diagrams* and *Sequence Diagrams*. COMPASS exploits tool's well-established extension mechanisms to extend traditional systems modelling as needed to model SoS. Starting from artefact created with the tool, COMPASS provide a well-defined denotational semantic of SysML blocks by means of the COMPASS modelling language (CML), a formal specification language that supports a variety of analysis techniques.

The DANSE methodology and tools are mainly based on the Unified Profile for DoDAF and MoDAF (UPDM). The latter has also been extended to cover the NATO Architecture Framework (NAF) and it provides more than fifty different model types grouped in eight viewpoints [6]. These viewpoints are: *Capability Viewpoint*, *Operational Viewpoints*, *Service Viewpoint*, *System Viewpoints*, *Service Viewpoint*, *Data & Information Viewpoint*, *Project Viewpoint* and *Standard Viewpoint*. In particular DANSE focuses on the six models that can be represented as executable forms of System Modelling Language (SysML).

In [7], the authors propose a formalism for relating basics SoS concepts by means of a UML class diagram. They identify as basic concepts *SystemType*, *System-Of-Systems*, *Goal*, *Role*, *Service*, *Requirement*, *Port*, *Requirement* and *Port*. Consequently, they adopted their defined formalism to instantiate an operative SoS by means of adopting canonical UML diagrams such as Sequence diagram. The behaviour of CS is formalized through Timed Automata and its dynamicity/evolution is achieved by means of Graph Grammars.

An example of modelling SoS by means of SysML is given in [8] where the authors exploit different diagrams and in particular executable diagram in order to simulate Net-centric SoS through the Petri Net formalism. In [9] the authors propose the use of SysML in representing an SoS in general and for a particular applicative scenario. They propose to adopt and in some cases to extend canonical SysML diagrams in order to model different aspects of an SoS. They defined concept Diagram as an extension of class diagrams to depict the top-level systems of an SoS and external stereotypes. This helps in identifying the boundaries between the system and its environment. They adopted the *class diagram* with an aggregator operator to represent that a component is

composed by a set of other components. They proposed the adoption of a *requirement diagram* with an additional stereotype, i.e., *critical requirement* which is a particular type of *requirement*. This diagram groups together requirements according to qualitative and quantities metrics to support a trade-off analysis. They adopt canonical *use case diagrams* to represent the set of action an SoS performs. The SysML *activity* and *sequence diagram* are exploited to represent the SoS at the functional level and its exchanges of messages, respectively. Finally, they exploit a *block diagram* as a refinement of their *concept diagram,* which aims at representing blocks/component with well-defined interfaces, i.e., *serviceports* and *flowports*.

The approach presented in [10] describes how several SysML models can be used to support a set of needs that the authors deemed essential for an SoS, namely *translating capability objectives*, *understanding systems and their relationships*, *monitoring* and *assessing changes*, *developing and evolving the SoS architecture, addressing requirements* and *solution options*. The authors propose to apply a Model-Driven Systems Development (MDSD) approach [11] to an SoS. The first step consists in determining capabilities and actors through *use cases diagrams* by defining what is in the system and what remains outside, as stated in a *context diagram*. Use cases determine the top-level service or capabilities and the major actions necessary to perform the use cases and all of the alternate actions. Finally, two different diagrams describe the interactions, i.e., *black box sequence diagram* and w*hite box sequence diagram.* Black box sequence diagrams show the flow of the requests that pass between the SoS and the environment while white box sequence diagrams depict the flow of requests between the constituent systems, and between the constituent systems and the external entities.

Among others, the approaches presented in this section show the utility of adopting SysML formalisms in order to model different architectural and non-architectural aspects of an SoS. This supports different types of analysis and it represents a first step towards executable artefacts, which can be automatically derived from SysML. As shown in this section, in the literature different attempts exist to apply SysML approaches to specific viewpoints that we deemed essential in providing architecture for Multi-Critical Agile Dependable and Evolutionary SoS. Nevertheless, an architectural framework that provides an integrated support to all these viewpoints is still missing. The architectural framework will benefit of the approaches proposed in the literature in supporting specific viewpoints (when they exist) and it will integrate SysML specific solutions to provide a usable high-level support for designers of SoS.

3   The AMADEOS Architecture Framework

The AMADEOS architectural framework (AF) is described by means of a high-level perspective of activities and artefacts involved in SoS design phases and by its viewpoint-based specialization.

3.1 High-Level View

The high-level representation of the AF is shown in Fig. 1 as a pyramid made of four different layers, namely *Mission, Conceptual, Logical* and *Implementation.* Apart from the *Mission* block, all the remaining levels are organized in slices, each corresponding to a specific viewpoint.

The starting point of the AF consists in defining the ***Mission*** of an SoS. The mission is commonly formalized by means of a document of intents created by enterprise managers having in mind a high-level perspective of the system and a clear definition of business-related issues. The document of intents is written in natural language to formalize the overall objectives and functionalities of an SoS starting from a shortened version of the glossary illustrating main SoS concepts and other related mission-relevant arguments.

At the ***Conceptual Level***, it is possible to consider a subset of viewpoints depending on the target SoS and its mission; however in AMADEOS we focus till to collaborative SoS, for which we identify a set of viewpoints that must be considered as mandatory. Inputs to these levels are the document of intents describing the mission, the conceptual model [12] defining main SoS concepts and their relationships and the AMADEOS meta-requirements [13], which can guide the identification of requirements for specific SoS instances. For each viewpoint (corresponding to a slice of the
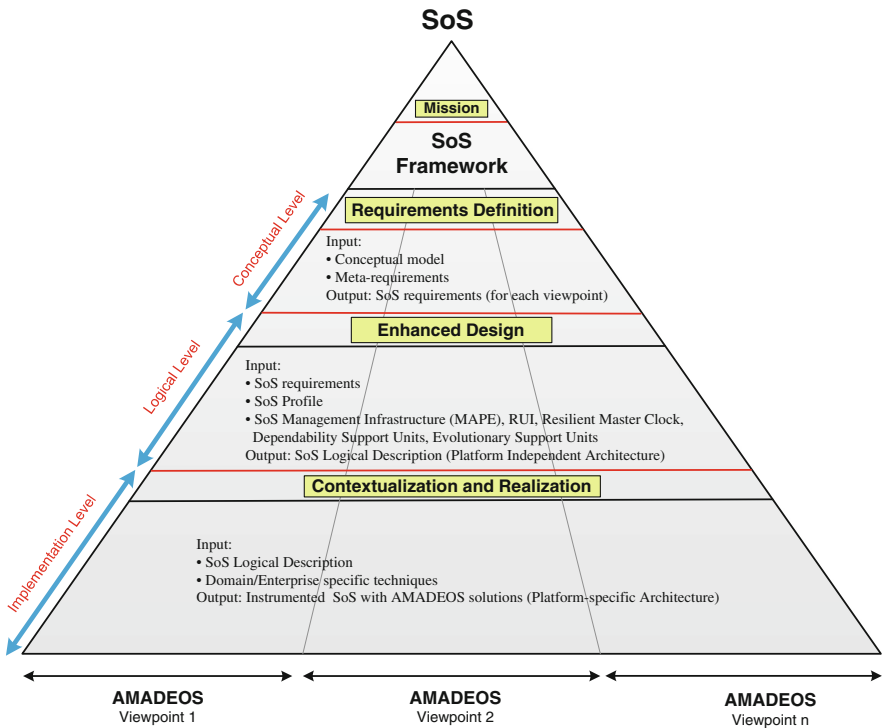

Fig. 1. AMADEOS architectural framework

pyramid), the SoS is examined and described. The resulting description should be the requirements of the SoS (these can be expressed in natural language, as well as using formalisms for the description of requirements). The identification of relations between the different viewpoints is carried out at this phase.

The *Logical Level* provides support for designing an SoS based on the viewpoints requirements in the AMADEOS SysML profileand the Building blocks defined in Sect. 4. The output of this phase consists in the platform independent description of the SoS in a semi-formal language (SysML), for the different viewpoints.

The *Implementation Level* leads to the integration of new CSs with already existing and deployed CSs starting from the logical architecture defined at the previous level and domain/enterprise specific techniques. At this level, the input logical architecture is refined and instrumented with domain/enterprise specific technologies which belong to the enterprise implementing the SoS instance.

We depict in Fig. 2 a process-based view of each level of the AF. We represent the basic task ad the input/output artefacts involved at each level. This gives a more detailed description of the relationships and evolution of the main artefacts produced in each level of the pyramid (see Fig. 1) and the relations between levels (through the top-down processes of refinement and instantiation, and bottom-up processes of generalization and abstraction). The artefacts categories at each level are intended to be generic enough to fit all the viewpoints.

On the *Mission Level*, a relatively slow-paced cycle takes place to address the continuous synchronization between the operational needs, the currently targeted
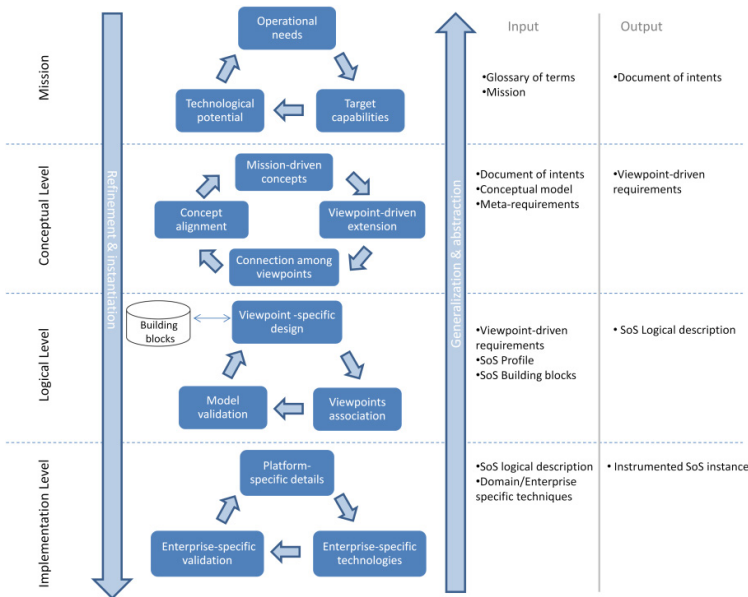

Fig. 2. Refinement and evolution of processes in AF design

capabilities of the SoS architecture and the technological possibilities to achieve the needs. At this stage, enterprise managers iterate the above phases to determine the mission of the SoS which is then formalized in a document of intents along with possible target solutions to be implemented.

On the *Conceptual Level* the alignment between the overall envisioned concepts and the SoS domain takes place more frequently. On this level the AMADEOS concepts which are relevant to achieve the mission are extracted from the document of intents and then filtered based on the viewpoint to which they belong. Further details may be added at each viewpoint descriptions to support the targeted capabilities within the SoS domain. Connections among viewpoints descriptions are identified in an early stage before similar concepts are aligned with each other, if needed.

On the *Logical Level*, cycles occur at a more rapid pace and are used to ensure that all desired functionalities and qualities are supported by the developed architecture. To this end, building blocks are selected and further integrated to obtain a design model which is generic enough to be applied to different types of platform. The design process follows a viewpoint-based perspective based on which target models are created for each viewpoint. Application specific details are added at this stage before viewpoint models are linked with each other according to the dependencies early identified at the *Conceptual level*. At *Logical level,* wrappers for legacy CSs have to be defined in terms of proper RUI interfaces which connect such legacy components to the rest of the logical SoS. Finally, validation activities take place, e.g., either by supporting the generation of models that are correct by construction or through predefined consistency checks. The generation of models, the integration of building blocks and the model consistency checks are made possible by exploiting the AMADEOS profile.

The most frequent cycles occur at the *Implementation Level*, where the SoS architecture is defined at its most fine-grained level by augmenting it with specific platform-dependent and specific technologies which are exploited by the target enterprise in order to obtain an operational SoS instance. At this stage the possibly available legacy components may be added to the platform-dependent architecture, provided that they have been properly encapsulated in the SoS at the logical level. This implementation model may then be validated through the technologies which are commonly adopted in place by the enterprise. In order for this phase to be supported, it is necessary that specific validation techniques adopted in the enterprise comply with the AMADEOS profile specification. However, it is not the main focus of AMADEOS to provide full support to implementation of single CSs. Nevertheless, this phase includes all the steps from the platform independent architecture to the architecture showing how each and every feature in the product should work and how every component should work. This phase is kept in the framework for completeness.

3.2   Viewpoint-Driven Analysis

The AF has been represented through a high level view which describes the processes of defining an operational SoS instance starting from the mission definition. The architectural viewpoints required for supporting this definition are the ones considered in the AMADEOS vision, i.e., structure, dynamicity, evolution, dependability and

security, time, multi-criticality, emergence. We describe in the following how the AF can support the activities required by each viewpoint.

Viewpoint of Structure. The Structure viewpoint concerns with representing the overall structure of the SoS. It focuses on architectural concerns of an SoS and it is closely related to other issues like SoS constraints, RUMI and semantics of communication. Indeed, defining interfaces among CSs is important as this stage to support their communications.

The input to the conceptual level of an SoS is the mission (or vision). This entails the overall objectives of the SoS as well as the required functionalities. The structure viewpoint entails examining these objectives and determining the constraints of the interfaces, and the communication, between constituent systems. Unlike the other viewpoints, the structure viewpoint places restrictions upon the activities of the SoS.

By addressing the meta-requirements in the context of the specific mission, a set of structural requirements can be identified that restrict the overall architecture that will be eventually delivered. For example, from [CONSTR 11], standards compliance of one or more CSs may be very important, particularly in use cases such as in the Smart Energy domain. Also the conceptual model is exploited as the vocabulary of concepts to be adopted.

The output of the conceptual level consists of a set of requirements that relate to the structural architecture of the SoS.

The logical design of an SoS architecture will be based upon the Structure requirements identified at the conceptual level and the building blocks identified along with the SysML profile. The SysML Block Definition Diagram (BDD) is used 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. A Block provides a unifying concept to describe the structure of an element or a system. This type of diagram helps a system designer to depict the static structure of an SoS in terms of its CSs systems and their possible relationships. By means of BDDs it will be possible to model the static structure of CSs, their interfaces and how the communication among CSs is achieved.

The output of the logical level will be a platform independent SoS architecture specification from a structural point of view. This will consist of the outline of the CSs identified by the requirements and the RUMIs that specify the interactions between these former CSs.

On the implementation level, the platform independent structural design from the enhanced design level is further concretized using specific contextual requirements, towards building the SoS structural architecture. For example, specific CSs may already exist and may need to be integrated. In the structural viewpoint, this will lead to specific RUMIs that are used to define how CSs will interact. These RUMIs consist of the communication protocols that define the messages that will be shared between CSs. The implementation level is very specific to the actual CSs involved and the operational context. The output consists in a fully contextualized SoS structural architecture.

Viewpoint of Dynamicity. Dynamicity refers to short-term changes in an SoS, which occur in response to changing environmental or operational parameters of the CSs. These changes can refer to offered services, built-in structure and interactions with

other entities, and may have different effects, such as SoS adaptation or the generation of emergent phenomena.

Starting from the SoS mission, the dynamicity requirements and the conceptual model, the output of the conceptual level is the set of dynamicity requirements, i.e., requirements related to the dynamicity viewpoint for the specific mission. The latter are the input of the logical level which also exploit the SysML profile and the building blocks to support the generate of the platform independent SoS architecture. The building blocks of the SoS management infrastructure defined are exploited to achieve dynamicity requirements, through the monitoring, analysis, planning and execution activities. Instantiation of the profile is connected with the Structure viewpoint of the SoS. Interactions elicited among CS take into account the service provided at the RUI interfaces as regulated by the SLA.

At the implementation level, the generic SoS architecture is instantiated into a platform-specific SoS architecture. This includes, among others, the implementation of RUIs that integrate monitoring and execution features that implement the MAPE-K architecture, and SLA-oriented reconfiguration operations. It results an architecture specialized by the enterprises with their adopted technologies to provide support to dynamicity though a platform-specific architecture.

Viewpoint of Evolution. Large scale Systems-of-Systems tend to be designed for a long period of usage during which the demands and constraints put on the system will usually change, as well as its environment. Evolution is the process of gradual and progressive change or development of an SoS, resulting from changes in its environment or in itself. In managed SoS evolution, the modification of the SoS keeps it relevant in face of an ever-changing environment; whereas in unmanaged SoS evolution, on-going modification of the SoS occurs as a result of on-going changes in (some of) its CSs.

At the conceptual level, starting from the mission, the evolution meta-requirements and the conceptual model, a set of evolution requirements produced. The latter are exploited along with the SysML profile and the building blocks at the logical level. In particular, instantiation of the profile is connected with the Structure viewpoint of the SoS. Interactions elicited among CSs take into account the service provided at the RUI interfaces, and the business value improved by evolution. The logical level results in the platform independent SoS architecture.

The role of the implementation level is to translate the generic SoS architecture into a platform-specific SoS architecture with, among others, evolution aspects. This includes RUI modification, and has a tight connection with the time viewpoint to ensure backward compatible evolution versions. Through the architecture as specialized by the enterprise it is possible to provide support to evolution though a platform-specific architecture.

Viewpoint of Dependability and Security. Dependability and security are essential properties of an SoS since they affect its availability, reliability, maintainability, safety data integrity, data privacy and confidentiality.

The conceptual level, build the set of dependability and security requirements from SoS mission, meta-requirements and the conceptual model. Dependability and security are important to ensure the proper functioning of an SoS. At the conceptual level, the input is the overall objectives and functionalities required to meet the mission of the SoS. Dependability and security requirements are not stand-alone requirements; they are connected to the other requirements, including time, multi-criticality, and others, that compose the set of requirements for the SoS.

At the logical level the Dependability and Security requirements are exploited to define the dependability and security components of the SysML profile There are two packages: "*SoS Dependability*" and "*SoS Security*". One of the key concepts in SoS dependability and security is splitting functionalities into well-defined components and interfaces such that the number of components that require explicit trust is kept to a minimum. In the context of the SysML profile, each block in the Block Definition Diagram has interaction points for *itoms* flowing in and outside the block. We first consider the functionalities required by the SoS and determine how security-critical each functionality is. We then consider what kinds of components make up the SoS and map functionalities to components. The most security-critical functionalities should be grouped together. Thus, the SoS will have a small number of highly-trusted, security-critical components. Less security-critical functionalities will be handled by less secure components. There will be different levels of dependability for each CS and different levels of security for each SoS.

The output of the logical phase is a platform-independent SoS architecture. The latter is exploited at the implementation level to create a platform-specific architecture specialized by the enterprise with their adopted technologies. For defining platform-specific trustworthy CS one could rely on the trustworthiness-enhancing design patterns described in the OPTET project [14]. This comprises a number of UML Patterns, which from the AMADEOS perspective can be seen as Dependability and Security architectural patterns.

Viewpoint of Time. Time does not only play an important role in the control of the physical environment of an SoS, where, for instance, the temporal properties of a control loop impact the efficiency and quality of control. It is also crucial for the information exchange between CSs, as in many cases timeouts and communication delays may decide whether the distinct CSs are able to serve their purposes. Correct handling of time enables the reduction of cognitive complexity required to design an SoS and facilitates the integration of new CSs into the system. On the other hand, undefined timing of communication between CSs might introduce unintended emergent effects.

Time meta-requirements along with related concepts and the SoS mission are exploited at the conceptual level to generate time dependent requirements. System components and functionalities sensitive to the progression of time need to be identified and the requirements on their temporal behavior have to be specified. This mainly comprises requirements on timeliness of interactions between CSs (e.g., the exchange of information to avoid collisions between cars has to take place before the cars collide), and the time synchronization of those CSs (e.g., requirements on the precision

of synchronization and time granularity). Since there is a close relation to other viewpoints, like Security, Dynamicity or Emergence, the temporal requirements have to be aligned with the requirements regarding the other viewpoints. Furthermore, the behavior of the SoS in case that some of the temporal requirements cannot be fulfilled has to be specified.

At the logical level the temporal behaviour of the SoS is designed based on the conceptual requirements defined in the level above. The mechanism to achieve a synchronized global time base among all CSs has to be defined (e.g., internal or external synchronization of time). Such a time base allows relating timestamps of different CSs with each other, and thus enables the temporal ordering of events in the SoS. The exact temporal interaction between individual types of CSs is modelled and included in the SysML RUMI specification. A precise temporal specification at this level simplifies the integration of CSs that have been individually designed and implemented at the next levels.

At the implementation level the producer of a CS brings the temporal specification of interactions between CSs into a real implementation using a specific platform. This includes implementing the time synchronization mechanism defined in order to achieve a common time base. As the implementation has to comply with the temporal model of interactions, unintended side effects of temporal misbehaviour are avoided, and hence, the integration of the CS into the SoS is simplified and the instrumented SoS instance is created.

Viewpoint of Multi-criticality. Multi-criticality supports the provision of services of an SoS with different criticality, such as safety-critical and non-safety-critical. Indeed, while some part of the SoS may have strong safety-critical requirements, other parts may be not so critical.

At the conceptual level the definition of multi-criticality requirements is carried out in order to support the definition of services with different criticality levels. To this end, the meta-requirements is exploited according to the SoS mission and using the related SoS concepts.

At the logical level the requirements along with building blocks and the profile are exploited to define the platform independent SoS instance. The SoS architecture and RUMI specification is done so that, recalling the macro-level of the general architecture of an SoS [13], CSs characterized by a specific criticality level n and a macro-level m can rely on CSs characterized by a criticality level greater or equal to their one owned by the same or a lower macro-level.

As stated by requirement [MULTI-CR6] a CS shall not rely on CSs characterized by a lower criticality level than its one. Thus, it is also necessary to have designed a clear architecture profile which details the structure of the SoS, detailing the interaction among the CSs. In this way it is possible to verify the correctness of the interaction among the CSs checking for violations of the aforementioned requirements. In the case that a CS offer several services that are characterized by different criticality levels, then a precise specification of the RUMI building block can help to preserve both the FCR and ECR, making failure propagation from non-critical services to critical one impossible.

At the implementation level the SoS Logical description (platform independent) is specialized by exploiting the enterprise-specific technologies based on specific enterprise technologies and it will result in a platform-specific instrumented SoS architecture and RUMI specification.

Viewpoint of Emergence. Emergence is an intrinsic property of the SoS and it concerns with novel phenomena that manifest at the macro-level (i.e., at SoS level) which are new with respect to the non-relational phenomena of any of its proper parts (i.e., CSs) at the micro level. The rationale behind emergence is that by composing CSs, either positive or detrimental global emergent phenomena may occur. Managing such phenomena can help avoiding unsafe unexpected situations generated from safe CSs, and may help eliciting positive emerging phenomena.

Appropriate effort shall be devoted to *monitoring*, *analysing* and *predicting* detrimental emergence phenomena and to *mitigating* (executing appropriate reactions) their effect on the SoS. For non-detrimental emergence, it is desirable, but not mandatory to monitor, analyse and predict emergence phenomena. Emergence may be influenced or generated by modifications to the Structure (e.g., adding new components which introduces new functionalities, or adding new components that may change the error model, e.g., introducing new Itoms which enables new interoperability between CSs), dynamicity and evolution (making the system able to make changes to the way its CSs interacts with each other and how the system is aligned with changing business requirements). Note that emergence phenomena may *cause violations* to handling of time, dependability and security of the SoS/CS.

In the Conceptual level, starting from the SoS mission, the meta-requirements and basic SoS concepts, we identify the instantiated emergence requirements.

The Logical Level concerns with applying the profile to identify emergence and categorize it according to the strength and predictability of effects. Because of the nature of the emergence concept, in we deemed not sufficient to simply elicit an emergent behaviour. We also consider worth capturing operational aspects related to emergence by considering an SoS in action. For these reasons, in we consider two possible diagrams to represent emergence through Block Definition Diagram and Sequence Diagram. The building blocks defined in Section Y support the monitoring, analysing, planning and executing mitigating activities required by the emergence management requirements. The instantiation of the profile should be tightly connected with the Structure viewpoint of the SoS. Interactions elicited among CS should be defined according to the *Request-Response* model and take into account the service provided at the *RUI interfaces* as regulated by the SLA. For supporting early identification and mitigation of emergence, particular attention has to be devoted to the interactions through stigmergic channels. The design process will also consider application and domain specific details which will be added by the designer. Finally, validation activities will check the correct application of building blocks, their integration and the usage of SoS domain specific concepts (possibly available through an SoS profile).

The platform independent architecture resulting from the logical is instantiated, configured and to linked the architectural elements, which support the achievement of the emergence requirements through the implementation of a platform-specific instance.

Evolutionary Aspects. A SoS evolves over time as constituent systems are modified, replaced or added, or due to its relevant environment (gradually) changes. This evolution is driven by incremental, new, and changing requirements of the SoS. An architectural framework for SoS should provide a tool aimed at predicting possible evolutionary paths based on anticipated requirements and use-cases.

Scenario-Based Reasoning for SoS Architecture Design. In architectural systems engineering the use of scenarios is not uncommon. It is a cost-effective means of controlling risk and to maintain system quality throughout the processes of software design, development and maintenance [15, 16] Preparing for evolution of an SoS, a scenario-based approach can also be adopted to guarantee that the development that an architecture undergoes is sensible, i.e. it must guarantee that the quality goals of the system are still met.

By using scenarios to guide the design of an SoS architecture, the context of the envisioned SoS is incorporated into the possible design choices by the architect. Established scenarios provide a narrative, which enables communication about future requirements and capabilities between different stakeholders [16]. Scenario-based design is a user-based approach in which different use-cases of a system are defined by narratives, from which a lower-level description of the system can be extracted. However, not every SoS can be described by narratives focused around use-cases and user interactions. Moreover, a narrative provides the intended use of a system from the perspective of a single expert or end-user, whereas in the context of SoS single use-cases are more related to the constituent systems than to the SoS as a compound structure. Therefore, a more methodical approach is needed, in which multiple experts can define relevant states and variables that may describe the possible evolution of the SoS and its relevant environment.

Scenario-based reasoning (SBR) [17] provides a methodical approach to generate and explore scenarios. In the SBR approach, scenarios are built from a set of variables, and each combination of variable states makes up a single scenario. Relevant scenario variables are those that influence the design of the system, such as variables that denote for example: environmental conditions, organizational dynamics, economic conditions, technological development, and interactions with the system form a user perspective. Such variables can have dependence relations between them which are, for example: causal, functional, influential, or probabilistic. For instance, enabling a certain security feature in the system will typically have an influence on its usability.

SBR enables what-if exploration to reason about possible future conditions and consequences for the architecture of an SoS. Through the analysis of different scenarios and their dependencies, inconsistencies can be revealed that may have consequences for the eventual architectural design of the system. Through the identification (also generation) of scenarios from a model describing the context under which the SoS will be deployed and the possible future uses of the system, evolving requirements may be elicited. By thinking about how to operationalize these requirements, insights are acquired about how they map to the architectural design of the system.

Figure 3 shows a small sample model from an environmental point of view, from which possible scenarios can be extracted for analysis. It depicts causal relations
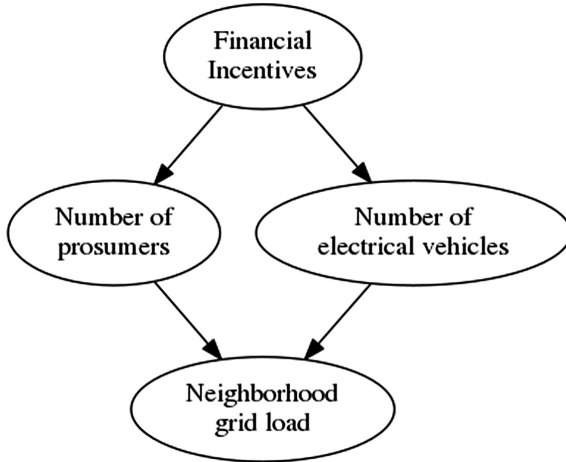
Fig. 3. A small example causal model for SBR

between the possibility of providing financial incentives for electrical vehicle use and energy production by consumers. Increased popularity of these use-cases in turn has an effect on the load placed on the local neighborhood grid.

4 The AMADEOS Building Blocks

In this section, we present the AMADEOS architectural building blocks which are exploited in the AMADEOS architectural framework.

4.1 SoS Management Infrastructure

The SoS management infrastructure in terms of a set of patterns which are applicable to enact monitoring, analysing, planning and execution strategies. The latter are developed as highly-dependable services, which we deemed essential for an SoS architecture. In order to implement the support to the above services we got inspired by the literature of Autonomic computing [18] which is a promising approach for a dependable architecture of very large information systems [3]. In particular, we propose to adopt the well-known MAPE-k cycle to implement the above services through Monitoring, Analyze, Plan and Execution components.

Our idea is to implement such patterns by means of composing CSs interacting with each other through well-defined RUI interfaces. These patterns are: (1) *Hierarchical Control*, (2) *Master/Slave*, (3) *Regional Planner*, (4) *Coordinated Control* and (5) *Information Sharing*. Patterns (1), (2) and (3) implement the so-called *Formal Hierarchy*, while patterns (4) and (5) implement the *Non-formal hierarchy.* We recall

that *Formal hierarchy* and *Non-formal hierarchy* have been discussed in Chap. 3 of this book.

Formal Hierarchy. In a Formal hierarchy any CS at level n is controlled by a CS at level $n + 1$. It follows that the MAPE components are placed in the CSs forming the controlling level, i.e., level n, while controlled CSs are placed at level $n - 1$. We consider three possible instances of this pattern as follows. The *Hierarchical Control* pattern consists in having a CS implementing all the MAPE phases (see Fig. 4).

In the *Master/Slave* pattern (see Fig. 5), the controller CS implements A and P, and then delegate to additional CSs M and E (Fig. 5).

In the *Regional Planner* (see Fig. 6) the controller CS implements only the Plan phase while it delegates to a set of CSs Analysis, Monitoring and Execute phases. The CS implementing the Plan phase operates for a region of CSs for which it is responsible.

Non-formal Hierarchy. In a Non-formal hierarchy CSs at level $n - 1$ interacts with the others at the same level by creating a whole at the level n. It follows that all controlled CSs and the CSs implementing the MAPE components are all placed at the same level, i.e., level $n - 1$. Two possible implementations are as it follows.
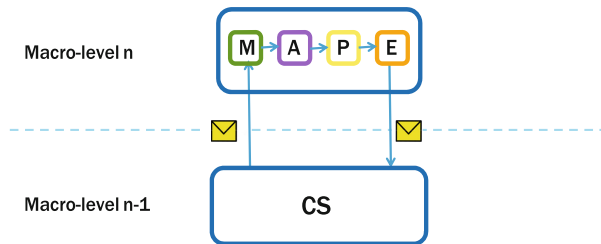
Fig. 4. Hierarchical control pattern
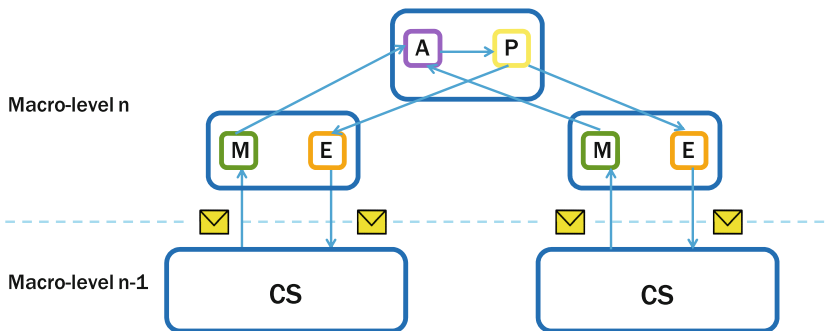
Fig. 5. Master/Slave pattern
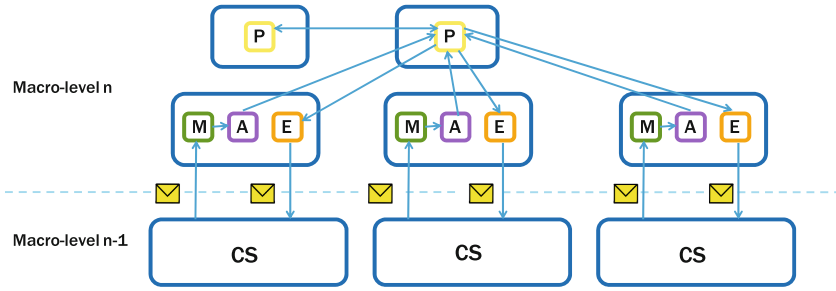
Fig. 6. Regional planner pattern

In the *Coordinated control* pattern (see Fig. 7) each of the CS at level *n* implements all the M, A, P and E phases. The latter coordinate their operation with corresponding peers of CSs at the same level (Fig. 7).

In the *Information Sharing* (see Fig. 8) is similar to the *Coordinated control* pattern but only interactions between Monitors are allowed.

Patterns composition. Each pattern presented in the earlier section exploited CSs at two possible abstraction levels. For the hierarchical control, we have at the higher level the *managing CSs* implementing the control of *managed CSs* which, in turn, have been represented as black boxes. For the holarchycal control, we have *managed* and *managing* CSs all at the same abstraction level, where all the *managed* elements are represented as black boxes, as well. The application of the above patterns may be applied compositionally and recursively by arbitrary replacing the managed CS by any other pattern.

Fig. 7. Coordinated control pattern

Macro-level n

Macro-level n-1
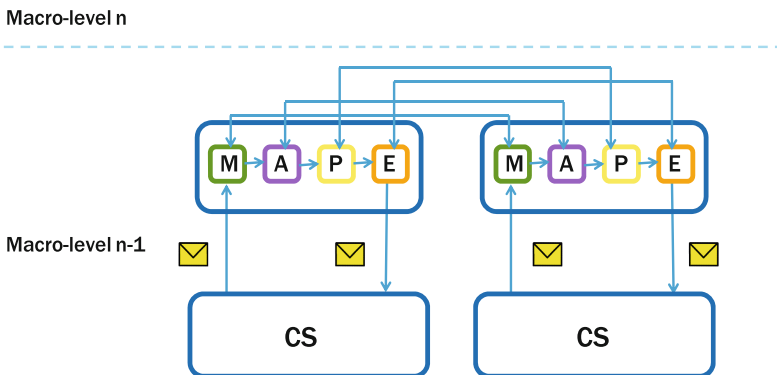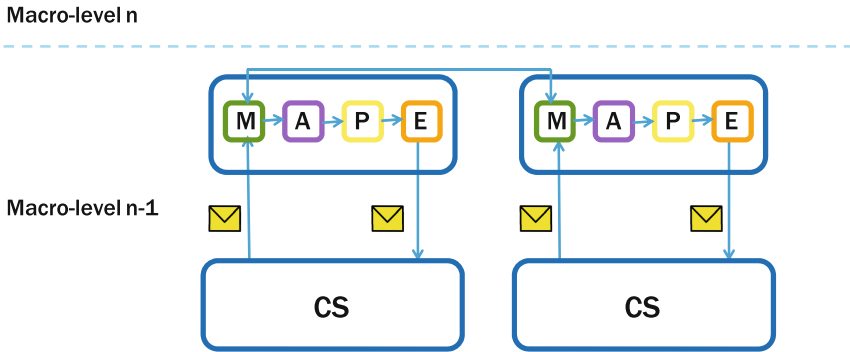
Fig. 8. Information sharing pattern

Finally, in addition to the presented patterns, a CS, being it a managing or a managed element, may interact with the physical environment by implementing the MAPE components. To this end, we introduce the atomic pattern as shown in Fig. 9.

Communication Infrastructure. The communication among the MAPE building blocks is achieved by appropriate interfaces whose nature depends on the objective of the communication, either physical entities or messages. Consistently with the AMADEOS conceptual model, we adopt RUMIs to support the communication among MAPE blocks for managing SoS, since we only require the exchange of information, i.e., Itoms, and not physical entities (which would require RUPIs). Indeed, in the presented management infrastructure, our MAPE blocks do not receive physical entities but simply messages, which can be sent/received within a single CS or across CSs. Those messages have been graphical represented in the pattern as yellow envelope items. The only exception is the atomic pattern, which supports the interaction with the physical environment and consequently it requires the adoption of RUPIs to exchange physical entities. Noteworthy, we only represent RUIs to support the communication of MAPE blocks, which span different CSs while we neglect to consider MAPE interactions within a single CS.
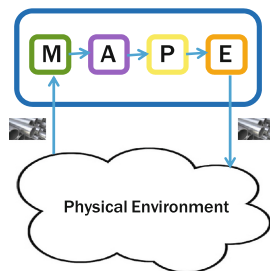
Fig. 9. Atomic pattern

4.2   Resilient Master Clock

Resilient master clock (RMC) is a resilient fail-silent master clock based on satellite-based time synchronization (e.g., GPS or Galileo signals), to provide a dependable global time base for cyber-physical Systems-of-Systems in AMADEOS.

5   The RMC Is Detailed in Chap. 6 of This Book. Supporting Facilities for AMADEOS

5.1   Introduction

The *supporting facility tool*[1] is used to model, validate, query, and simulate an AMADEOS based SoS using the Blockly tool[2]. Blockly is an open source library for building visual programming editor or a visual DSL (domain specific language). Blockly has been adopted to ease the design of SoS by means of simpler and intuitive user interface; thus requiring minimal technology expertise and support for the SoS designer. Its main features are: (i) Fast, and only a modern web browser is required; (ii) Intuitive and simpler user interface; (iii) Easily extendable with custom blocks; (iv) Ability to check constraints at design time (user defined and pre-defined constraints) and warn user when the user makes mistakes; and (iv) Support code and XML generation.

The supporting facility tool is a generic SoS designer in accordance with the AMADEOS conceptual model and for this the Blockly tool has been customized to be used for SoS modelling. The flow of model-driven engineering using the supporting facility tool is depicted in the Fig. 10. The SysML meta-model is first transformed to Blockly blocks. These blocks could be used in the supporting facility tool to create an SoS model.

The main motivation of *supporting facility tool* is: the current SoS design tools are complex and non-intuitive for general SoS designers; also, many of the existing tools expect designers to be well-versed with object-oriented concepts. The goal of supporting facility tool is to simplify and provide means to rapid modelling of SoS using the SysML profile (meta-model). In traditional modelling environment, large models have been known to be difficult to design and maintain; and often leading to spaghetti diagrams. The tool aims to reduce the complexity by using collapsed views instead of lines to connect blocks. Also, the tool aims to warn user of common errors/mistakes during modelling and helps in quicker testing of SoS through simulation. The main advantage of using the supporting facility tool is that the SoS designer need not have deep knowledge of SysML/UML; the tool hides all the object-oriented concepts from the user and provides full compliance with the AMADEOS profile. The only prerequisite is high-level knowledge about the profile and knowledge of the supporting facility tool usage.

---

[1] http://blockly4sos.resiltech.com.

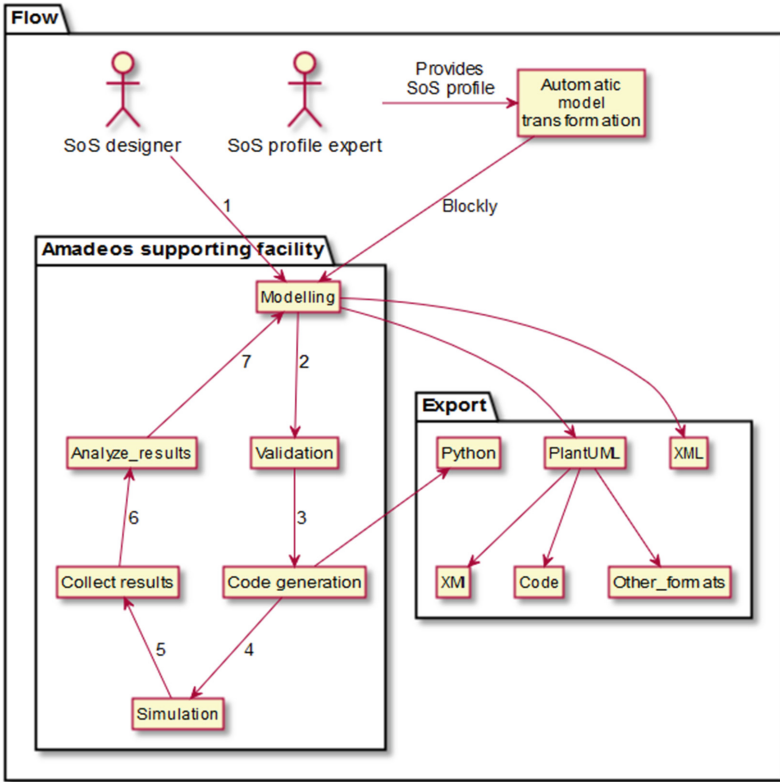[2] https://developers.google.com/blockly/.

Fig. 10. Flow of MDE using the supporting facility tool

The supporting facility simplifies the task of SoS modelling by reducing the pre-requisites to start modelling. Once the supporting facility is installed on a web server, it can be accessed from any machine using a modern web-browser. It can also be used locally without the need of a web server. It provides rapid modelling, validating, code-generation, and simulation facilities to the user. The supporting facility can generate three outputs: (i) the model in XML, (ii) Python code-generated for the simulation, and (iii) PlantUML version of the model.

PlantUML is a simple text based UML format which can be readily integrated with many tools[3]. The exported model in PlantUML may be used for further refinement or formal analysis. For example: the PlantUML model can be *viewed* in Eclipse using plug-ins[4]. Though, full interpretability between tools is an ongoing research topic and is under investigation.

---

[3] http://plantuml.com/running.html.

[4] http://plantuml.com/eclipse.html.
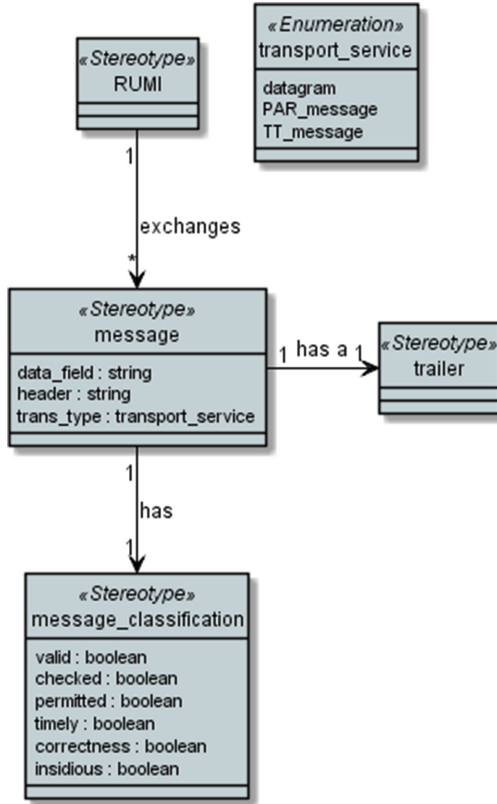
Fig. 11. An example subset of SysML meta-model to be transformed to Blockly

Python is a general purpose portable language, and the Python code generated by the tool can be further refined and also be used to connect to other simulators or external systems for interaction while running simulation.

As the supporting facility tool is based on the SysML profile (the meta-model) derived from the AMADEOS conceptual model, the SysML (in XML) is transformed into Blockly by using PlantUML as an intermediary language. PlantUML is chosen as an intermediate format as it is a simple text format which makes debugging during the model transformation easier. Below is an example of model transformation from SysML in Papyrus/Eclipse to Blockly (Figs. 11 and 12).

5.2   Modelling SoS

When the tool is launched, it creates a default SoS block called "*example_block*" as an example. All the blocks required to build an SoS can be found in the toolbox on left hand side. These blocks are imported from the AMADEOS SysML profile provided by a profile expert. Each block in the tool contains information taken from the AMADEOS
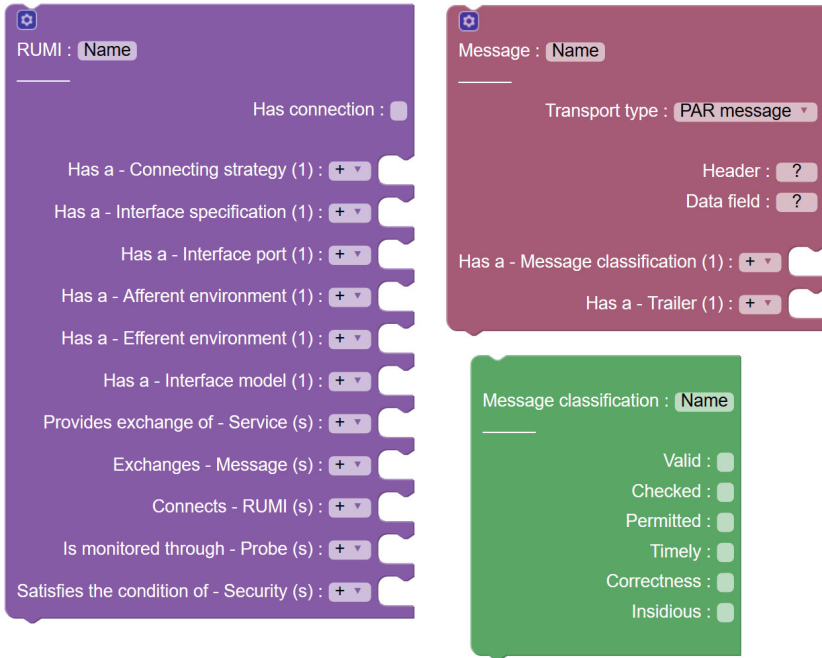
Fig. 12. SysML (Fig. 11) imported to Blockly

conceptual model to guide the SoS designer. For example, help for CS block can be found by right clicking a block and selecting *Help*. Also, each imported block in Blockly is associated with a viewpoint/building-block, for example all blocks associated with Communication viewpoint is present in the Communication category in the toolbox.

Traditionally, Blockly requires users to drag and drop blocks from flyout/toolbox to create new blocks. To improve usability and correctness, a Blockly API*: Blockly. FieldDropdown()* is used to show the list of blocks compatible to be connected for a given block; this lets the user create blocks in an easier way. Figure 13 shows an example, where to add a *Technique* block, the tool shows that the following new compatible blocks can be added: "*Fault forecast*", "*Fault prevention*", "*Fault removal*", and "*Fault tolerance*". In the profile, Technique is an abstract block and the above four blocks inherit the Technique block.

A block once created can have three views, (i) collapsed view, (ii) partially-collapsed view, and (iii) uncollapsed view as shown in Fig. 14. Collapsed view allows the user to reduce the number of blocks screen on the screen and to focus on the current editing block. Partially collapsed block only shows the non-empty attributes of a block hence the designer may choose to view only the attributes defined. Full view/uncollapsed view is used to see all the attributes of a block. A user can cycle between the three views by double-clicking the block.
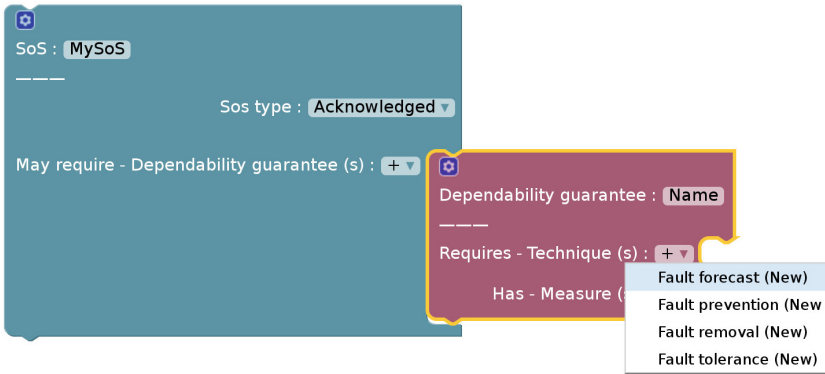
Fig. 13. Aiding user to add new blocks through dropdown

Also, for each block it is possible to see the attributes related to selected viewpoints/building-blocks as shown in Figs. 15 and 16. This is achieved by providing a mutator button for each block at top left hand side.

To provide an intuitive modeling environment, the supporting facility uses a readily available open source plug-in called *Type-Indicator*[5]. This plugin indicates all the blocks compatable (with yellow color) with current block while it is being dragged, as shown in Fig. 17 (the block *cs4 is* currently being dragged*).

Requirements Management. Requirements management is an important aspect of an SoS design, where traceability of requirements must be viewed/monitored. Requirements may be divided based on the viewpoints and building-blocks: Architecture, Communication, Dependability, etc. Each block maintains the list of requirements it meets and each requirement block maintains the list of blocks which satisfy it; thus offering full traceability (Figs. 18, 19, and 20). Blockly also supports adding comments to blocks to make the design clearer.

Constraints in the Model. Each block exports a list of variables in *JavaScript* which can be used to define constraints. These variables are defined in the format: *block. <relation_name>_<block_type>* (For e.g. a CS block exports block.provides_service). Also, each block exports shortcut variables in the form *block.m_<block_type>* (e.g. for a CS block, block.m_service). Instead of *"block"* keyword, a shortcut variable *"b"* may also be used.

For multiple inputs, a dictionary variable in the form *"d_<variable_name>"* is also exported. This variable is used to access variables by using block name as a key (e.g. for an SoS, Using the variable *block.d_cs['cs1']* the CS in the SoS having name *cs1* can be referred. Constraints make a model precise, the constraints provided by the tool uses JavaScript's **"eval"** function to evaluate the constraints and change the color of block to *black* in colour if the constraint is not satisfied. The constraints are evaluated at

---

[5] https://github.com/HendrikD/blockly-plugins/tree/master/type-indicator.
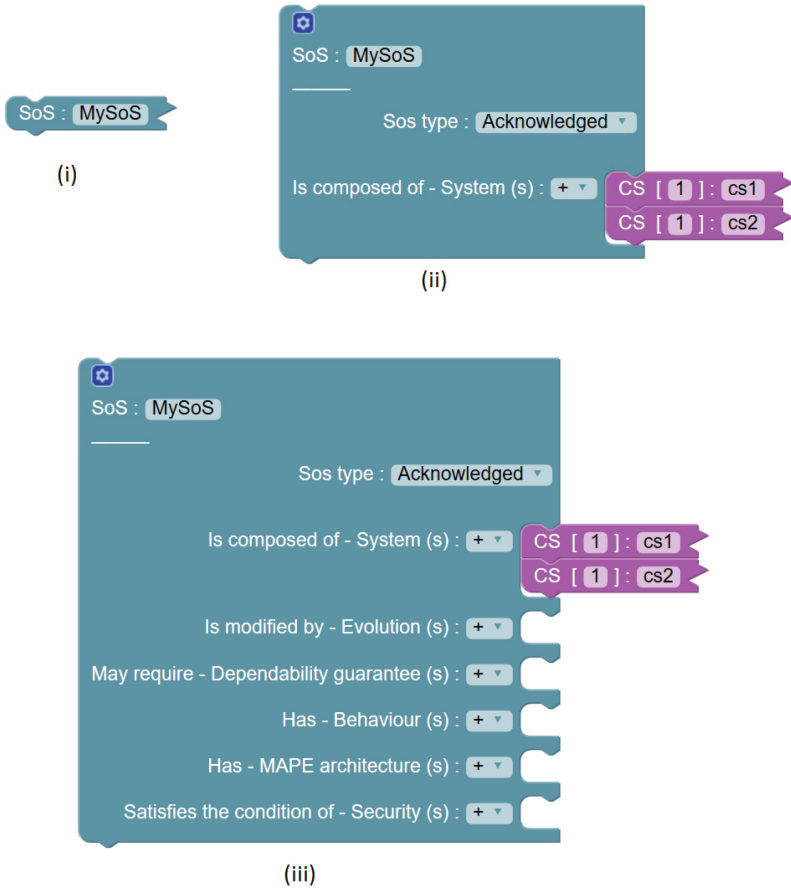
Fig. 14. Three ways to view a block (i. Collapsed, ii. Partially-collapsed, and iii. Un-collapsed)

each *onchange* event of block. Constraints rely on the variables exported by a block. Figure 21 shows an example use of constraints.

Constraints may also be used to detect causal loops which may lead to emergence scenario in SoS (Fig. 22).

Model Querying. On large models it is difficult to visualize the entire SoS, and then the need for custom viewpoints arises. Blockly does not use lines to show relationship between blocks and uses collapsed views to hide the complexity of an SoS model. Model querying can be used search for blocks which satisfy a given condition (using a query). It may also be used to visualize a model in traditional view (i.e. showing blocks and its relationship with other blocks using lines). To query a given model, a user can right click on workspace and choose "*show query diagram*. In the query diagram, user may write a filter function for querying the model. For example, *return true;* indicates that no filtering is required (i.e.: show all blocks for the model depicted in Fig. 23); which results in the graph as shown in Fig. 24. Using the filter "*return b.
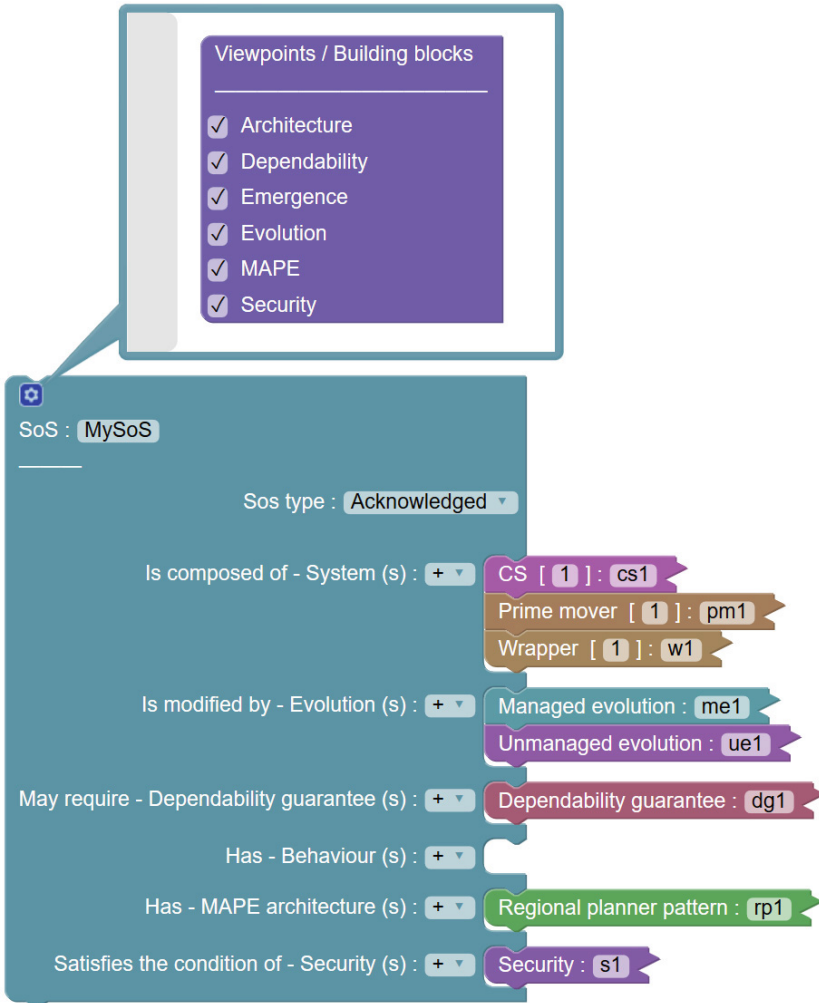
Fig. 15. Viewpoints/building-blocks of a block can be enabled or disabled

of_type == 'RUMI';" which indicates to highlight all blocks of type *"RUMI"*, this query returns the graph depicted in Fig. 25 (note that *b* is a shortcut for variable *block*). Model querying helps in visualizing custom viewpoints of SoS and can be helpful in identifying issues in the SoS design.

Adding a Link to a Block. One way to design a SoS is by using links to existing blocks. Creating links can help reuse an existing block; however, this is different from copy-pasting a block in blockly. Links are reference to the linked blocks. For example: CSs can be created on workspace and only links may be added to the SoS block, as shown in Fig. 26.
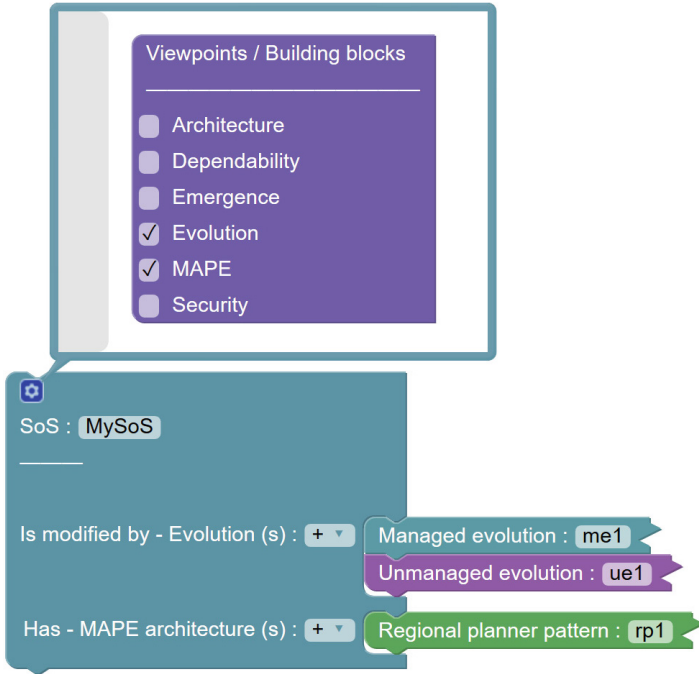
Fig. 16. Filtered view of SoS
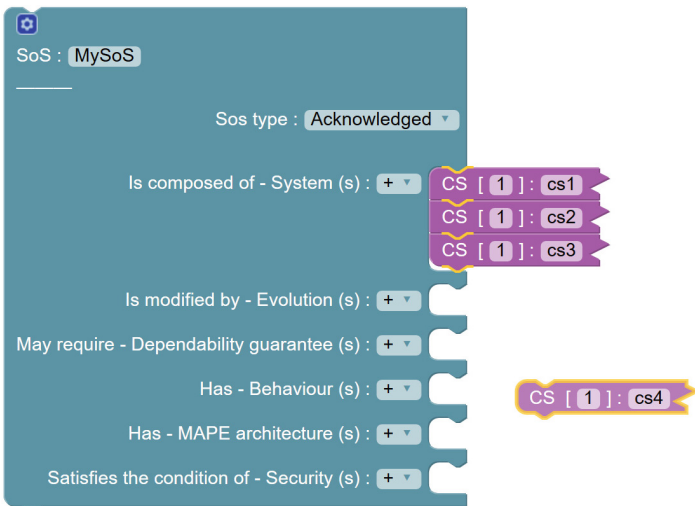
Fig. 17. Use of Type-Indicator Plug-in (compatible connections for cs4 are indicated by yellow colour) (Color figure online)
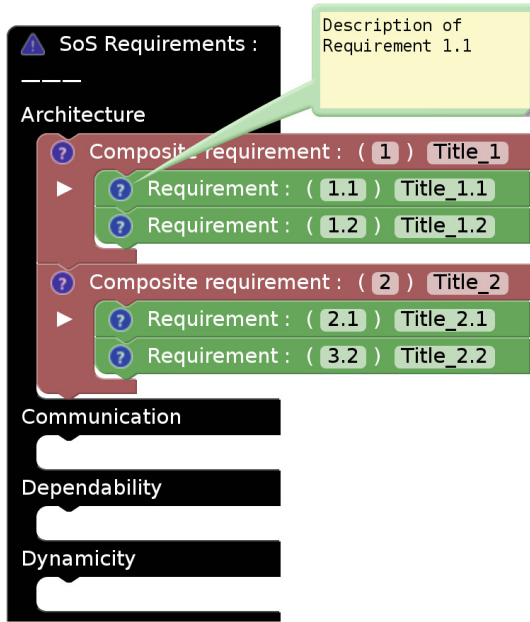
Fig. 18. Example of blocks related to requirements management
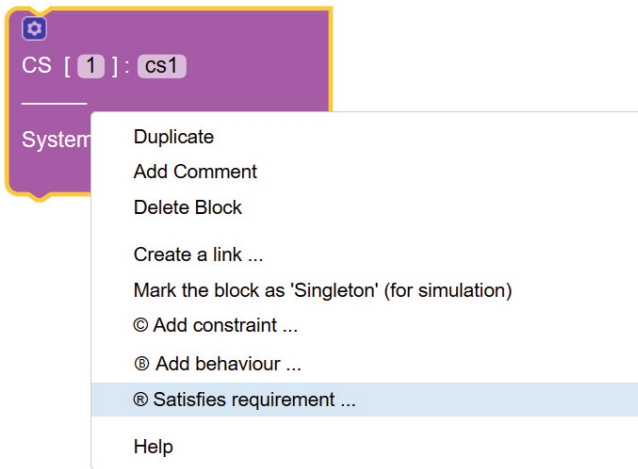
Fig. 19. Each block can satisfy a requirement (by providing the requirement ID it satisfies)
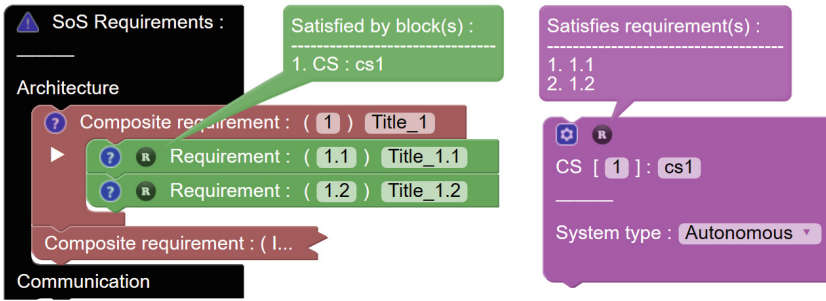
Fig. 20. Traceability of requirements
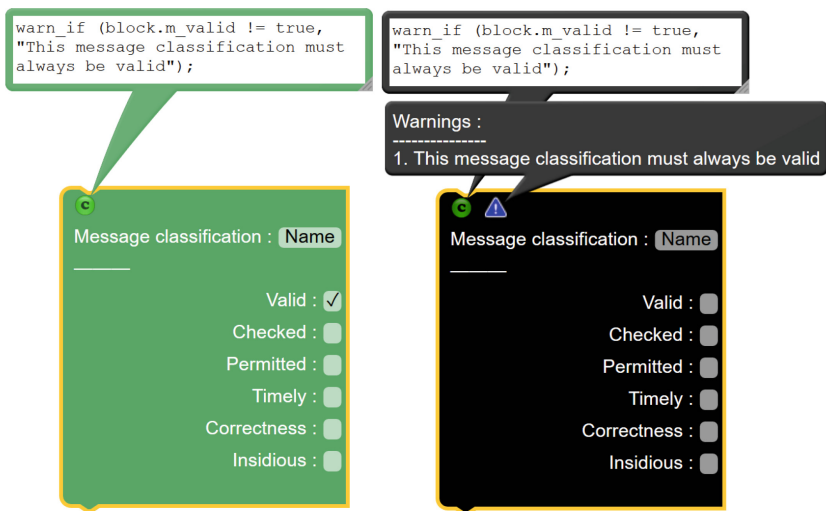
Fig. 21. An example of a constraint where the member variable m_valid is checked

Grouping for Modular SoS Design. The supporting facility allows grouping of compatible blocks together to modularize the design. For example, all CSs can be grouped together as shown in Fig. 27. The group block helps in organizing the model into meaningful groups. Also, when a block of a group blocks is refered, the group name is indicated to distinguish it from other blocks which may have similar names.

5.3   Simulation Environment for SoS

Behaviour. Once a static model is defined, behaviours may be added to any block. To add a behavior, the user can right click on the interested block, and choose "Add behavior". The behavior represents the code to be executed during simulation, and can be written in Python programming language (as shown in Fig. 28). The function names
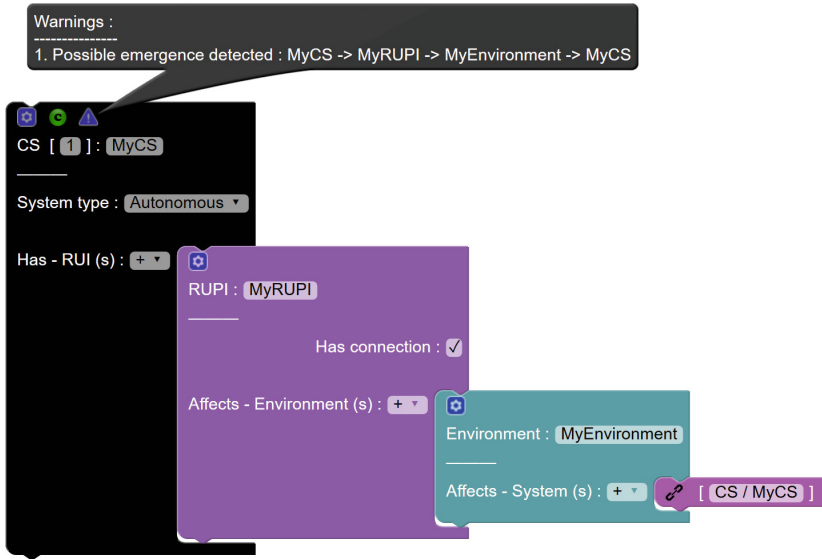
Fig. 22. Detecting emergence in model through constraints
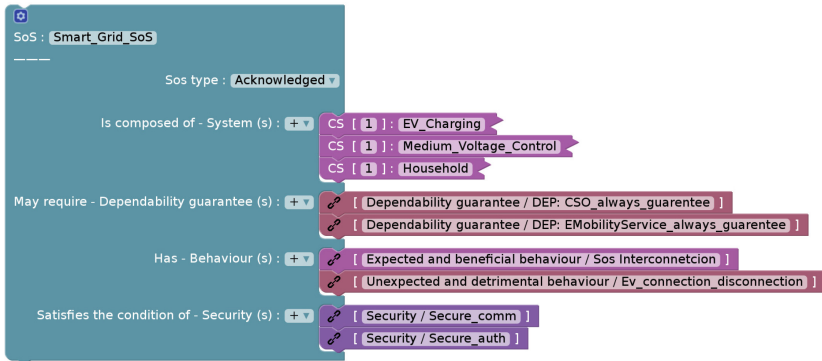
Fig. 23. Model querying large models (for query "return true;" i.e. show all blocks)

init, *start*, and *run* can be defined and are executed during initialization, start of the block, and during the course of simulation respectively.

The *run* function for a service block has a special meaning and is exposed as a TCP/IP server. All the behavior code written for all blocks are integrated in to a single file for code generation.

XML and Code Generation. After the model is loaded, it can be exported to XML and code for simulation by clicking on the appropriate buttons on the top right hand side of
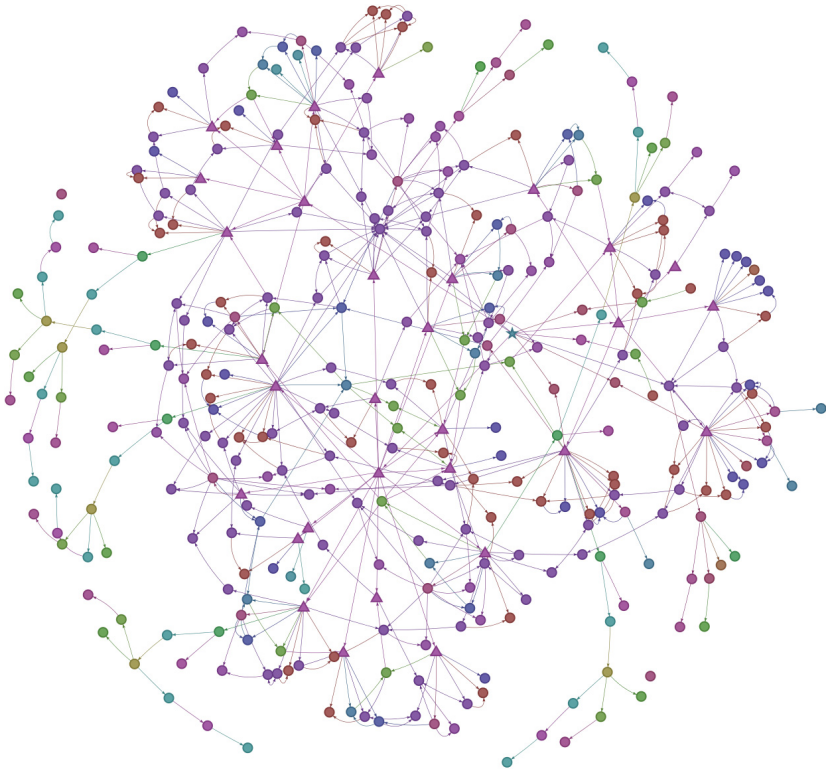
Fig. 24. Result of "return true;" query

the tool. Unique object names are generated for all blocks in a format: *<block-type>_<block-name>_<block-id>*.

Simulator Components. The simulator is a set of Python programs meant for executing the desired scenarios created by designer (the scenarios may also be represented using sequence-diagrams). The simulator consists of the following main components: Object initializer, Registry, Sequence diagram, GUI, Runtime sequence diagram, **log** generator, and Clock.

Object Initializer. The simulation initializes each object/block defined in the model using the block's constructor. Single inputs are considered as strings/integers/object; whereas multiple inputs are considered as array. If a value for single input is not provided, its value is considered as *None* in Python; whereas for multiple inputs it is considered as an empty array *[]*.

Certain blocks such as CS/Wrapper/Roleplayer/CPS can have a member called *"cardinality"* (Fig. 29). It indicates number of objects to be simulated. This is implemented by using *copy.deepcopy ()* function of Python on the original object. Each instance is assigned a *_instance_id* (1 to N); where *N* is the cardinality specified in the
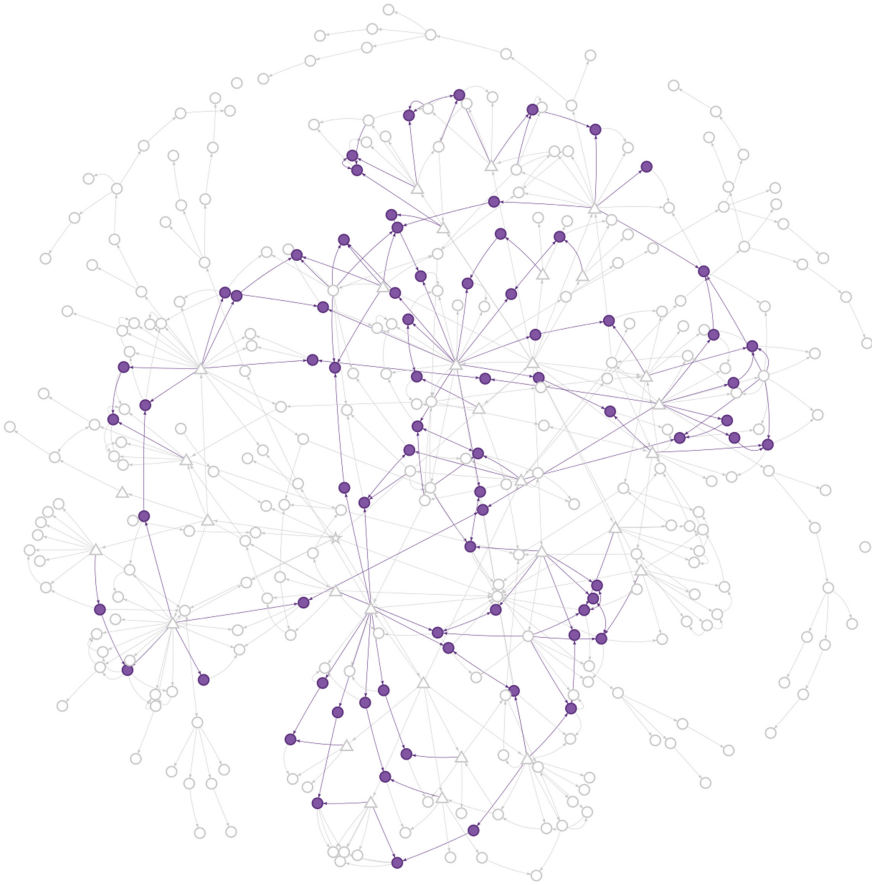
Fig. 25. Result of "return b.of_type == 'RUMI';" query (select all RUMIs)

model. Example: the below model creates an SoS called MySoS and has 200 CSs having name cs1. Each of the CSs will have *_instance_id* attribute from 1 to 200.

Registry. Registry is one of the main components of the simulator. It is a service that maintains the list of services offered by various CSs registered in a SoS. It is used by the CSs to search for a particular service. In the simulator, the registry is implemented as a TCP/IP server, where CSs can add/remove/update their own service information. Having a known common registry allows the possibility to run the simulation across several computer systems connected together.

Sequence Diagram. Blockly blocks related to sequence diagrams helps to create non-ambiguous sequence diagrams, which can be readily converted to code. Simulator follows the exact sequence as defined in the sequence-diagram created by the user. Thus, the code generated from the sequence diagram (Fig. 30) is executed right after the simulator has been started and initialized. A sequence diagram is added to model to simulate a scenario (Fig. 30); the sequence diagram designed in supporting facility tool
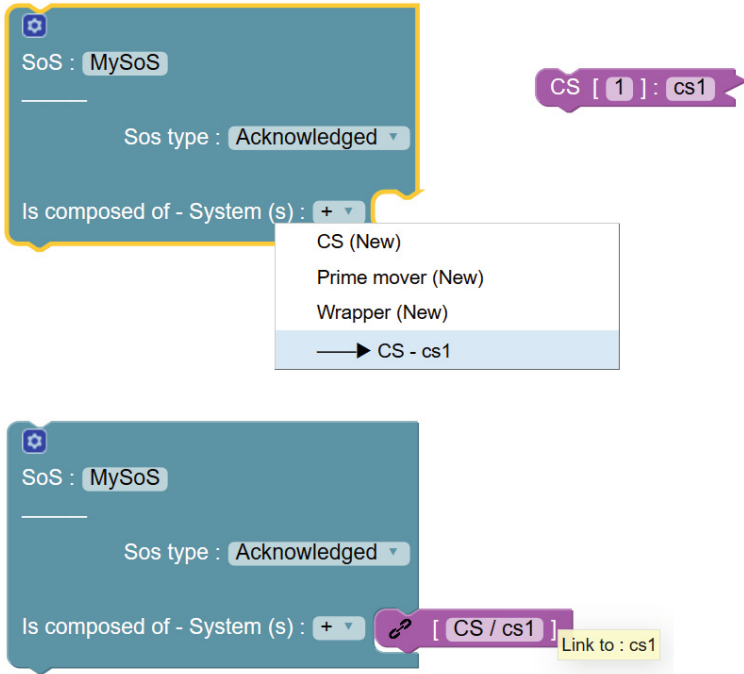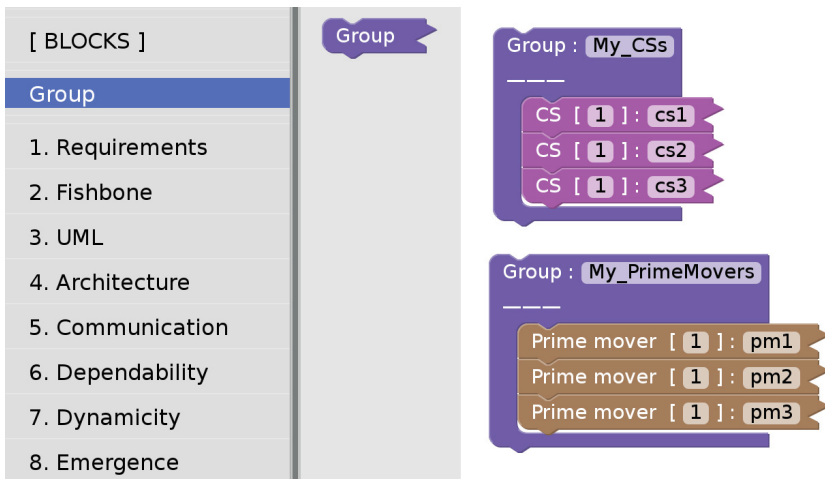
Fig. 26. Reusing an existing block (cs1) using links.

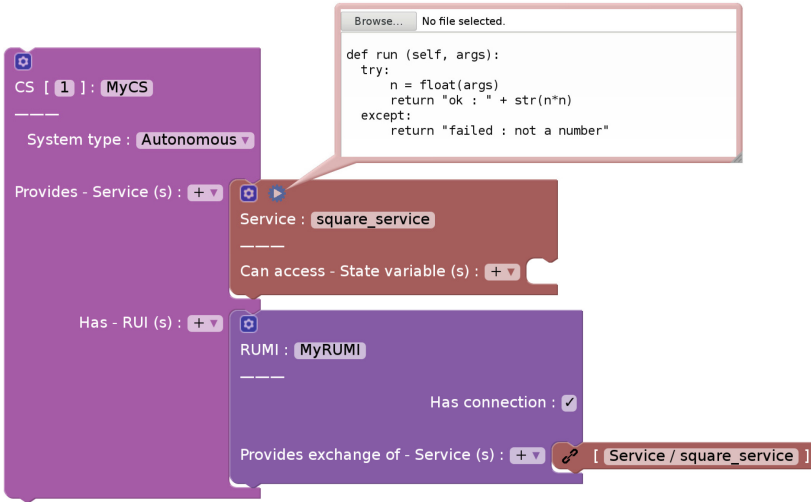**Fig. 27.** Similar blocks can be grouped together

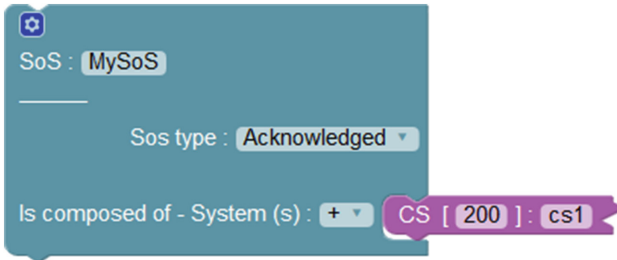**Fig. 28.** Example of behaviour for a service

Fig. 29. Specifying the cardinality for CS – cs1

can also be visualized in classical sequence by right clicking the sequence diagram block and selecting "load sequence diagram". This loads the sequence diagram in sequence diagram window, which can be viewed by right clicking workspace and selecting "*Show sequence diagram*".

GUI. The GUI of the simulator is the starting point of the simulator, and it lets the user select the systems to be run on the current machine and displays the progress of the simulation by logging activities performed by blocks (such as CS/RUMI, etc.).

Runtime Sequence Diagrams. Given the sequence diagram created by the user, the simulator starts executing the sequence diagram. While executing, each activity performed by RUIs are logged as sequence diagram in PlantUML format by adding timestamp to each activity. This creates a runtime-sequence diagram *(in result.seq file)*, which shows what actions have occurred with its timestamp. The runtime-sequence diagram also shows the delay between each action.

Fig. 30. Sequence diagram in supporting facility tool using Blockly

Log Generator. The logs generated by the simulator can be saved in a file and can be used to compute the metrics of interest. These metrics may indicate the quality of SoS by measuring performance/delays/failures etc.

Clock. The simulator uses the system clock of the machine on which the simulation is running. However, it is possible to setup an experimental setup in which, each CS runs on different machine using different clocks. These clocks could be synchronized with a master clock e.g. the RMC developed in task D 4.4 [19]. Faulty scenarios (regarding time synchronization) are also possible to generate by perturbating local clocks of each machine, or by removing master clock from the network.

Simulator Code Organization. The code generation of the supporting facility tool generates a *".zip"* file in the format *"<model-name>.zip"*, which contains the complete code for the simulation. The simulator code is created for each model based on the specified sequence diagram. The generated code when extracted is organized as shown below (the model name is *"sos-model"*):

The top directory name is in the format *"SoS-Simulation-<Date-and-Time>"*, which hosts two executable code files: *"simulation-on-unix.sh"* and *"simulation-on-windows.bat"*; both are meant to start simulation on UNIX-based and Windows-based machines respectively. The *model-<Date-and-Time>.xml* file consist of model in XML format.

The *"src"* folder contains the simulator code, the constructor code for each blocks, and initialization code of the block objects created in the model. The entire code consists of the following files:

(1) **amadeos.py**

This file contains all the constructors for each block defined in the SoS profile from the conceptual model. This file may be edited to add/refine additional generic classic functionalities.

(2) **model_behaviour.py**

This file contains the behaviors for each block defined during the SoS modelling. The behaviours are associated with each instance of a block and not for each class. Thus the behaviour for one object will not be shared by other objects of the same class.

(3) **sos.py**

This is the main simulator code which is started by *"simulation-on-windows.bat"* or *"simulation-on-unix.sh"* file. This code sets a random seed for random number generation, creates the registry, sets global data for simulation, and starts the user interface for simulation.

Also, this file contains code that starts the simulation by starting all systems as a thread, runs the code related to the sequence diagram, and waits for all threads to join.

(4) **sos_gui.py**

This file contains the GUI code for selecting the systems to be started on the running machine. Also, this file contains code for showing the log of activities performed by each CS.

Running the Simulation. After the code generation, a user can start the simulation by launching the file *"simulation-on-unix.sh"* or *"simulation-on-windows.bat"* (Fig. 31). When the simulation starts, a GUI is shown that allows the user to select the list of systems to be started, and the registry IP address and port. An example GUI is shown in Fig. 32. After selecting the list of systems to be started, the user can start simulation by clicking the *"Start simulation"* button.

Simulation Over a Network of Computers. The SoS simulation may also be performed over a network of computers. This is achieved by maintaining a common registry machine; thus forming a distributed system running various AMADEOS based systems in each of the computer systems communicating through TCP/IP.

This also allows the possibility of the simulator to interact with real legacy-systems. Each machine can run a set of systems (CSs/Wrappers/Primemovers). When run, a
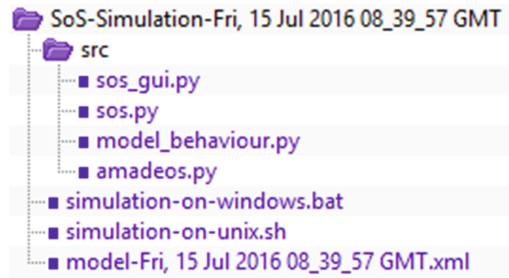
Fig. 31. Simulator code directory structure

separate thread is created for each selected system; and each system initializes itself and starts its RUIs in separate threads.

Example Run of an SoS Simulation. This section describes and the steps for running the simulation using an example simulation of a SoS model designed in the supporting facility tool. An example SoS may be launched from the dropdown found on the top left hand side of the tool.

5.4   Prerequisite to Run Simulation

As simulator code is written in Python 2.7, the pre-requisite to run simulation is an installation of Python version 2.7[6]. On Windows it is preferred to install _Python_ at c: \Python27, which is the default option provided by the installer.

5.5   Starting the Simulator

As mentioned earlier, the simulator code will be generated as a file in the form _"<sos-name>"_.zip, containing a _src_ folder and two files: _"simulation-on-windows.bat"_ and _"simulation-on-unix.sh"_.

The simulator starts when the user runs the script _"simulation-on-windows.bat"_ or _"simulation-on-unix.sh"_ depending on the operating system.

For security reasons, on some versions of Windows it may be required to right click on the _"simulation-on-windows.bat",_ click to properties, and check "Unblock" this file.

When the simulator starts, it shows a GUI (Fig. 32), where the user can specify the list of systems to be started on the current machine. After system selection, the user may click "Start simulator" to start the simulation. The user can see the log of activities appearing during the simulation on the GUI (Fig. 32).

After the simulation run, the user may close the GUI. The simulator generates the result of the current simulation, i.e., message passing between RUMIs and interactions between RUPIs - as a run-time sequence diagram in a file called _"result.seq"_.

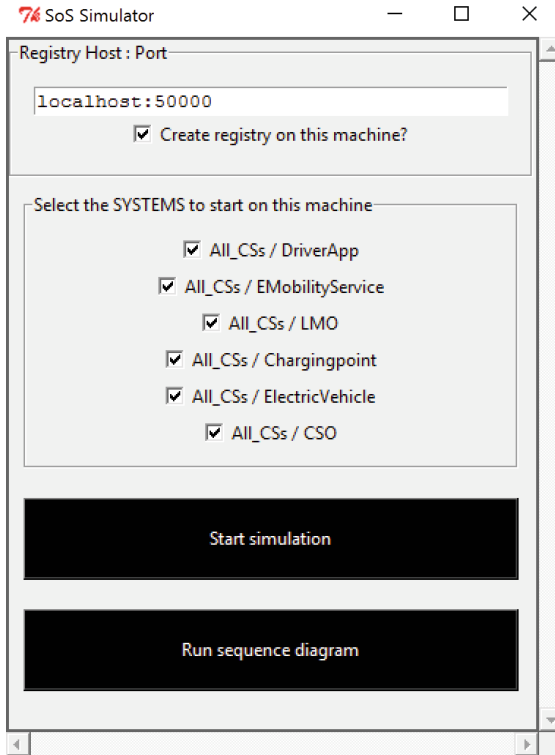[6] https://www.python.org/download/releases/2.7/.

Fig. 32. Start-up GUI of simulator

The result is saved in the same directory where the simulation was run. This file can be viewed by a PlantUML viewer or a sequence diagram-viewer available in the supporting facility tool by right clicking on workspace and selecting "*Show sequence diagram*". The sequence diagram frame can be extended to fit the page, and the user can use the "*Browse button*" in sequence diagram frame to load the *"result.seq"* file.

6   Conclusion

This chapter has introduced the architectural framework and supporting facility tools for AMADEOS based SoS.

This chapter has showcased the features of the supporting facility tool, focusing on simplicity and intuitiveness in modeling and simulating an SoS. The supporting facility tool also demonstrates the possibility of: design, validation, querying, simulation of system of systems. Case studies using the supporting facility tool will be presented in Chap. 8.

References

1. Schonenborg, R., Bieler, T.M.A., Fijneman, M.: System of systems architecture in ESA's concurrent design facility. In: SECESA - System and Concurrent Engineering for Space Applications, Lausanne (2010)
2. Muller, G., Van de Laar, P.: Researching Reference Architectures. Springer, Netherlands (2009)
3. Murer, S., Bonati, B., Furrer, F.J.: Managed Evolution: A strategy for very large information systems. Springer, Heidelberg (2011)
4. Atego: Artisan Studio, March 2012. http://www.atego.com/products/artisan-studio/
5. DelCOMPASS: Guidelines for Architectural Modelling of SoS. Technical Note Number: D21.5a Version: 1.0, September 2014. http://www.compass-research.eu/Project/Deliverables/D21.5a.pdf
6. delDANSE: DANSE Methodology V2 - D_4.3. https://www.danse-ip.eu/home/images/deliverables/danse_d4.3_methodology_v2.pdf
7. Gezgin, T., Etzien, C., Henkler, S., Rettberg, A.: Towards a rigorous modeling formalism for systems of systems. In: IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (2012)
8. 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), 203–220 (2008)
9. Huynh, T.V., Osmundson, J.S.: A systems engineering methodology for analyzing systems of systems using the System Modelling Language (SysML), Department of Systems Engineering, Naval Postgraduate School, Monterey (2006)
10. Lane, J.A., Bohn, T.B.: Using SysML modeling to understand and evolve systems of systems. Syst. Eng. **16**(1), 87–98 (2013)
11. Bohn, T., Nolan, B., Brown, B., Balmelli, L., Wahli, U.: Model driven systems development with rational products, IBM Redbooks (2008). http://www.redbooks.ibm.com/abstracts/SG247368.html?Open
12. Project AMADEOS: Deliverable D2.2 "AMADEOS conceptual model" (2015)
13. Project AMADEOS: Deliverable D1.1 "SoSs, commonalities and Requirements" (2014)
14. OPTET: OPerational Trustworthiness Enabling Technologies. FP7-ICT-2011.1.4 - Trustworthy ICT project
15. Kazman, R., Carrière, S., Woods, S.: Toward a discipline of scenario-based architectural engineering. Ann. Softw. Eng. **9**(1–4), 5–33 (2000)
16. Rosson, M.B., Carroll, J.M.: Scenario-based design. In: Jacko, J., Sears, A. (eds.) The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications. Lawrence Erlbaum Associates, pp. 1032–1050 (2002)
17. Conrado, C., de Oude, P.: Scenario-based reasoning and probabilistic models for decision support. In: 2014 17th International Conference on Information Fusion (FUSION) (2014)
18. Kephart, J., Chess, D.: The vision of autonomic computing. IEEE Comput. **1**(36), 41–50 (2003)
19. Project AMADEOS: Deliverable 4.4 - Design of Resilient Master Clock (2016)