Bridging software engineering gaps towards system of systems development

Marcelo Augusto Ramos

Bridging software engineering gaps towards system of systems development

Marcelo Augusto Ramos

Advisor: **Profa. Dra. Rosana Teresinha Vaccare Braga**
Co-Advisor: **Prof. Dr. Paulo Cesar Masiero**

Doctoral dissertation submitted to the *Instituto de Ciências Matemáticas e de Computação* - ICMC-USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computacional Mathematics. FINAL VERSION.

USP – São Carlos
July 2014

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

Ramos, Marcelo A.
R175b Bridging Software Engineering Gaps Tawards System of Systems Development / Marcelo A. Ramos; orientador Rosana T. V. Braga; co-orientador Paulo C. Masiero. -- São Carlos, 2014.
134 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2014.

1. Software Engineering. 2. Software Product Line. 3. Reengineering. 4. Statecharts. 5. System of Systems. I. Braga, Rosana T. V., orient. II. Masiero, Paulo C., co-orient. III. Título.

Preenchendo lacunas da engenharia de software rumo ao desenvolvimento de sistema de sistemas

Marcelo Augusto Ramos

I dedicate this thesis to God for giving me health, peace, prosperity and happiness;
To my father Waster (*in memoriam*) for teaching me to never give up my dreams;
To my mother Zeneide (*in memoriam*) for dedicating her life to my well-being;
To my aunt Wamir for having me as a son;
To my wife Adriana for loving me unconditionally;
To my children Juliana and Gabriel for their affection.

Acknowledgments

I thank to my advisor Profª Drª Rosana T. V. Braga and to my co-advisor Prof. Dr. Paulo C. Masiero for extending their hands to me in the most difficult moments of this journey, for the hours they devoted revising my work, for the countless emails they replied thoughtfully, and for sharing their professional experience. I extend the same thanks to Profª Drª Rosângela A. D. Penteado who have been advising me since my first step into the post-graduation program and gently accepted to collaborate on this thesis. Finally, I thank all those people who directly or indirectly contributed to this important achievement.

To all of you, my sincere THANK YOU!

Summary

Abstract 1
Chapter 1 3
Introduction 3
1.1. Context 3
1.2. Motivation 6
1.3. Objectives 8
1.4. Organization of the thesis 9
Chapter 2 11
Background 11
2.1. Initial considerations 11
2.2. System of Systems (SoS) 11
2.2.1. SoS definitions 12
2.2.2. SoS distinguishing characteristics 13
2.2.3. SoS classification 15
2.2.4. Family of Systems (FoS) 16
2.2.5. SoS engineering (SoSE) 17
2.2.6. SoS composition 18
2.2.7. SoS modeling 20
2.3. Software Product Line (SPL) 21
2.3.1. Definitions 22
2.3.2. General SPL Engineering (SPLE) process 22
2.3.3. Dynamic SPL (DSPL) 24
2.3.4. SPLE and SoSE integration challenges 26
2.4. Service orientation 27
2.4.1. The WS-* standards 28
2.5. Statecharts 30
2.6. Final considerations 32
Chapter 3 33
A generic SoSE process 33
3.1. Initial considerations 33
3.2. The proposed SoSE process 33
3.3. The current work 36
3.4. An example 37
3.5. Related work 40
3.6. Final considerations 40
Chapter 4 43
A requirement engineering approach to SoS 43
4.1. Initial considerations 43
4.2. Related work 45
4.3. The proposed scene-based RE approach 47
4.3.1. Overview of the approach 47
4.3.2. Example 48
4.3.3. The scene-based approach and the proposed SoSE process 50
4.4. Case study – Describing the sample virtual SoS 51
4.4.1. High-Level requirements – SoS 52

4.4.2. Middle-Level requirements – Scenes 58
4.5. Final considerations 66
Chapter 5 69
A modeling approach to SoS 69
5.1. Initial considerations 69
5.2. The proposed statechart extensions 70
5.2.1. Statechart extensions 74
5.2.2. Statechart extensions applied to systems 74
5.2.3. Statechart extensions applied to events 76
5.2.4. Statechart extensions applied to states 81
5.3. Modeling systems' interactions 82
5.4. Related work 88
5.5. Case study – Modeling the sample virtual SoS 91
5.6. Final considerations 96
Chapter 6 99
An SPLE extension towards SoS development 99
6.1. Initial considerations 99
6.2. The proposed SPLE extention 100
6.3. Case study – Composing the sample virtual SoS with SPL instances 102
6.4. Final considerations 108
Chapter 7 111
A reengineering approach extension to SoS 111
7.1. Initial considerations 111
7.2. Related work 112
7.3. The proposed reengineering approach extension 113
7.4. Case study – Composing the sample virtual SoS with legacy systems 115
7.5. Final considerations 119
Chapter 8 121
Conclusions 121
8.1. Summary 121
8.2. Contributions 121
8.3. Constraints 122
8.4. Remarks and Future work 123
Addendum 125
The sample SoS 125
The calculator 125
The publisher 125
The processor 125
Member systems 126
SoS characteristics 126
The resulting SoS 126
References 129

Figures

Figure 1. The GEO System of Systems (Shibasaki and Pearlman, 2009) 4
Figure 2. The Draganflyer X6 UAV (Draganfly, 2014) 5
Figure 3. The Draganflyer X6 equipped with payloads (Draganfly, 2014) 5
Figure 4. The Draganflyer product line (Draganfly, 2014) 5
Figure 5. System and System of Systems (Smith II, 2006) 12
Figure 6. FoS of cameras 16
Figure 7. Different SoS supported by the FoS of cameras 17
Figure 8. The U.S. Integrated Ocean Observing System (IOOS, 2013) 19
Figure 9. Service-oriented SoSE approach (Lewis et al., 2011) 20
Figure 10. SoS dynamism (Zhou et al. 2011) 21
Figure 11. Roles in a product line organization (McGregor, 2004) 22
Figure 12. General process for SPLE (Ziadi, Jézéquel, and Fondement, 2003) 23
Figure 13. SPL effectiveness (Clements and Northrop, 2001) 23
Figure 14. Home robot feature model (Lee and Kang, 2006) 25
Figure 15. Distinct configurations of the context-aware DSPL for smart homes: a) user at home e b) user out of home (Cetina et al, 2009) 25
Figure 16. WS-Discovery protocol (ad hoc mode) (WS-Discovery, 2009) 29
Figure 17. WS-Discovery protocol (managed mode) (WS-Discovery, 2009) 29
Figure 18. Hypergraph and Euler/Venn diagram 30
Figure 19. Higraph with Cartesian product (Harel, 1998) 31
Figure 20. Similar representations of a 5-clique (Harel, 1998) 31
Figure 21. Statechar's broadcast mechanism (Harel, 1987) 32
Figure 22. Ovelapping states (Harel, 1987) 32
Figure 23. The proposed SoSE process 34
Figure 24. The proposed SoSE process applied to the military domain 38
Figure 25. Competition scenario (Roberts e Walker, 2010) 39
Figure 26. Coalition US-Sing. SoS (Adapted from Huynh and Osmundson, 2006) 45
Figure 27. SoS concept and context diagrams (Huynh and Osmundson, 2006) 46
Figure 28. Scene-based RE approach 48
Figure 29. SoS scenes modeled as states 49
Figure 30. SoS feature model 50
Figure 31. Artifacts of the High-Level Requirements - SoS phase 50
Figure 32. Artifacts of the Middle-Level requirements phase 51
Figure 33. Sample SoS feature model 54
Figure 34. A use case of the sample SoS 55
Figure 35. The sample SoS use case diagram 55
Figure 36. Scenes of the sample SoS 58
Figure 37. Scene 1 – Compose/Enter SoS 59
Figure 38. Scene 2- Compose/Leave SoS 61
Figure 39. Scene 4 - Do Shared Calculation 62
Figure 40. Scene 5 – Publish Shared Information 63
Figure 41. SoS composition sequence diagram 65
Figure 42. View of a canonical complex system (Jennings, 2001) 69
Figure 43. Transitions and interactions in the SoS context 71
Figure 44. Phone call statechart (Harel, 1987) 72
Figure 45. Multi-layer printed circuit board 73

Figure 46. Using OCL to describe overlapping systems 75
Figure 47. Example of symbolic notations attached to systems 76
Figure 48. Broadcast, multicast and unicast interactions 78
Figure 49. Solving ambiguous situations 78
Figure 50. Example of symbolic notations attached to events 79
Figure 51. Unclustering the dial activity 80
Figure 52. Example of symbolic notations attached to states 82
Figure 53. Skype™-like system use cases 83
Figure 54. PCB-statechart of a Skype™-like system 85
Figure 55. Use case: *Make a Call* 87
Figure 56. Layered architecture 87
Figure 57. Modeling SoS as a Communicating Structure (Kotov, 1997) 89
Figure 58. Requirements for a Room Control [13] 90
Figure 59. Modeling relationships among entities (Tian et al, 2011) 91
Figure 60. Sample SoS feature model updated with location and video deliverables 92
Figure 61. Sample SoS PCB-Statechart 93
Figure 62. SPLE extensions 101
Figure 63. Two different SPL instances running as independent systems 103
Figure 64. Two different SPL instances running as SoS members 104
Figure 65. Two different SPL instances interacting in the SoS environment 108
Figure 66. Reengineering approach to SPL (Ramos and Penteado, 2008) 112
Figure 67. Instantiation of SPL products (Ramos and Penteado, 2008) 113
Figure 68. Reengineering approach extension 114
Figure 69. POS terminal and its usual HW peripherals (Verifone®, 2013) 116
Figure 70. POS terminal feature model 116
Figure 71. POS-App architectural model 117
Figure 72. POS-App new architectural model 118
Figure 73. System#1 – Calculator + Processor 127
Figure 74. System#2 – Calculator + Publisher 127

Tables

Table 1. SoS distinguishing characteristics (Boardman and Sauser, 2006) 13
Table 2. Paradoxes of SoS (Sauser, Boardman, and Gorod, 2009) 14
Table 3. Traditional x SoS engineering (Keating et al., 2003) 18
Table 4. Statechart extensions attached to systems 74
Table 5. Statechart extensions attached to events 77
Table 6. Description of Skype™-like functions 84
Table 7. Display feature Connection Map (CMap) 118

Abstract

While there is a growing recognition of the importance of System of Systems (SoS), there is still little agreement on just what they are or on by what principles they should be constructed. Actually, there are numerous SoS definitions in the literature. The difficulty in specifying what are the constituent systems, what they are supposed to do, and how they are going to do it frequently lead SoS initiatives to complete failures. Guided by a sample SoS that comprises all the distinguishing SoS characteristics and a generic SoS Engineering (SoSE) process, this thesis explores the SoS development from different Software Engineering (SE) perspectives that include requirements, analysis, design, and reengineering. For the Requirements Engineering (RE), we propose a scene-based RE approach to describe the SoS progressively as an arrangement of elementary but meaningful related behaviors named 'scenes'. The objective is making easier the description and the understanding of the SoS dynamism. For the analysis, we propose extensions to statecharts to visually improve the modeling of systems' interactions. They are symbolic notations that result from an analogy with multi-layer Printed Circuit Boards (PCB). The resulting diagrams are named PCB-statecharts. For the design, we propose an extension to the conventional SPLE process in such a way that SPL can become a natural source of SoS members. Domain engineering is extended to deliver components able to share abilities in SoS environments. Then, application engineers can design families of products that comply with different SoS requirements and still improve their products using the abilities of other SoS members. For the reengineering, we propose an approach extension to evolve legacy systems to SPL and then to SoS members. We demonstrate that when legacy systems are reengineered properly, they can share useful abilities, work cooperatively, and compose SoS.

Chapter 1

Introduction

1.1. Context

In the 90's the increasing complexity of certain solutions faced the software computer engineers with the need of designing multiple integrated systems. In response to this emerging paradigm several System of Systems (SoS) definitions have been proposed. Some SoS definitions compiled by Jamshidi (2009) are presented in the following.

- SoS are large-scale concurrent systems that are comprised of complex systems.
- In relation to joint warfighting, an SoS is concerned with interoperability and synergism of command control, computers, communications, and information and intelligence, surveillance, and reconnaissance systems. Primary focus: information superiority.
- SoS are large geographically distributed assemblages developed using centrally directed development efforts in which the component system and its integration are deliberately, and centrally planned for a particular purpose.
- An array system (SoS) is a large widespread collection or network of systems functioning together to achieve a common purpose.
- An SoS is a set of collaborative integrated systems that possess two additional properties: operational and managerial independence of the components.
- An SoS is a set of different systems so connected or related as to produce results unachievable by the individual systems alone.

The term SoS has been used consensually in the literature to describe compositions of large-scale often complex independent systems and their information-based relationships (Jamshidi, 2009; SEI, 2014). An example is the Global Earth Observation System of Systems (GEOSS, 2013) illustrated in Figure 1. The purpose of the GEOSS is to achieve comprehensive, coordinated, and sustained observations of the Earth system to meet the need for timely, quality long-term global information as a basis for sound decision making, initially in nine societal benefit areas: disaster, health, energy, climate, water, weather, ecosystems, agriculture, and biodiversity (Shibasaki and Pearlman, 2009). This is a typical application of

SoS present in the literature. Independent systems are composed to achieve goals that a unique system would rarely be able to achieve.

Figure 1. The GEO System of Systems (Shibasaki and Pearlman, 2009)

The above consensus about SoS makes researchers to focus almost exclusively on problems intrinsic to the composition of large-scale complex systems, e.g., interoperability (Morris et al., 2004; Smith II, 2006) and manageability (Maier, 1998; Meyers et al., 2006).

In some circumstances, however, the composition of conventional systems forming an SoS could ease the solution of problems currently handled by single systems. An example is the project of unmanned vehicles like the Unmanned Aerial Vehicle (UAV) of Figure 2.

Frequently, the vehicle and its payloads, e.g., GPS and cameras, are assembled to build a single system. In this case, they maintain part-whole relationships as exemplified in Figure 3. This may decrease the scalability, diversity, and flexibility of the solution. For example, the system may need to be fully rebuilt whenever existing payloads change or new payloads are added. Moreover, this may require changes in the system architecture, models and code. On the other hand, if the vehicle and payloads are independent systems designed to interact, they can be composed to achieve the same goals probably with less effort. Indeed, different payloads can be composed to achieve different purposes in a more flexible way.

Figure 2. The Draganflyer X6 UAV (Draganfly, 2014)

Figure 3. The Draganflyer X6 equipped with payloads (Draganfly, 2014)

In the above scenario, the UAV may not be able to accomplish different SoS goals. Indeed, requirements like flight autonomy and payload capacity may vary from an SoS goal to another. Product Lines (PL) would be a solution for this problem. Similar UAVs with distinct features could be instantiated from the PL to meet different SoS requirements (Figure 4), This could probably reduce costs and efforts associated to SoS projects.

Draganflyer Guardian Draganflyer X6 Draganflyer X4-P Draganflyer X4-ES

Figure 4. The Draganflyer product line (Draganfly, 2014)

Unfortunately, composing independent systems is not like assembling Legos™. Brownsword et al (2006) states that the difficulty in specifying what are the constituent systems, what they are supposed to do, and how they are going to do it often lead SoS

initiatives to complete failures. Defining and modeling systems' interactions comprehensively can increase the success rate of SoS initiatives. However, this is a challenge. The resulting models should remain clean and understandable even for an increasing number of systems and interactions.

This thesis is about SoS engineering (SoSE), its activities, and the challenge of composing conventional systems to deploy SoS. By composing we mean describing roles as well as designing interaction and collaboration rules so that member systems can improve their own purposes and still contribute collectively to the SoS goal. This thesis is also about conventional software engineering (SE) and the challenge of evolving existing approaches to support SoSE. Product lines complement the set of addressed areas because of their promising contribution to SoS, as introduced above.

1.2. Motivation

Currently, there are several problems to be solved before conventional systems can compose SoS naturally. From requirements to deployment processes, conventional software engineering (SE) methods, methodologies, and techniques would require improvements to support SoSE. For example, requirements should not describe a single system and its characteristics. Instead, they should describe interaction and collaboration rules able to support a set of roles to be played by the member systems towards the achievement of the SoS goal. The resulting environment should be attractive to candidate systems that may decide to participate of the SoS in a voluntary manner based on cost-benefit decisions. Member systems should be able to interact freely to create a diversity of relationships able to maintain the SoS stability in a long term. Over time, more systems can join the SoS to expand its capability while others can simply leave. Thus, relationships are supposed to be formed and vanished conveniently. This dynamism needs the SoS requirements to be able to evolve together with the mutable SoS boundaries. Actually, some requirements will only be known later during the SoS operation.

Describing the above scenario is currently a challenge to conventional requirements engineering (RE) processes used to foresee most of the system's characteristics and behaviors, and consider them static over time or subject to minor future changes. These processes should be evolved or new ones should be proposed so that SoS can be properly described.

Supposing the interaction and collaboration rules as well as the roles to be played were comprehensively and correctly described, another problem takes place for SoS engineers, i.e.,

modeling the relationships among member systems. Complex tasks may be carried out by a large number of systems and may comprise even a larger number of interactions. In this case, conventional SE techniques used to represent systems side-by-side and connect source and target interacting points through arrows may become complex very fast and a puzzle over time. Examples include UML sequence and communication diagrams (Booch, Rumbaugh, and Jacobson, 1999). Yet, this is not the only problem. The diversity of relationships enables the same task to be carried out by different systems in different numbers over time. This way, a task, the involved members and their relationships can hardly be modeled statically as usual. Finally, the SoS goal can comprise a large number of related tasks to be modeled what makes the SoS modeling process even more complex.

Modeling the above scenario is currently a challenge to conventional SE modeling techniques used to foresee most of the interacting components and their relationships, and consider them static over time, e.g., component-based development (Heineman and Councill, 2001). These techniques should be evolved or new ones should be proposed so that SoS goals can be properly modeled in terms of dynamic relationships among member systems.

Supposing systems' interactions were properly modeled, another problem takes place for SoS engineers, i.e., today's systems are mostly not ready for composing SoS without great effort. Although the evolution and miniaturization of the hardware and the creation of communication standards enabled a countless number of network-enabled computer systems to connect and exchange information, conventional SE processes have not evolved together with the hardware in such a way that they can support the SoS paradigm in a natural manner. Indeed, several SE development processes have been proposed over time. Mostly, they focus on common critical points to deploy a single end product with fixed requirements and well-defined boundaries (Keating et al., 2003). As discussed above, this is not enough to deploy an SoS. To make that possible, conventional SE processes, methods and techniques should evolve so that composition is taken into account since the very beginning of the development process. Few examples are given in the following.

Software Product Line (SPL) (Clements and Northrop, 2001), and Service Orientation (Erl, 2008) are examples of SE methodologies that should evolve to support SoSE. Services can support platform-independent interaction mechanisms like dynamic compositions, discovery processes, and information exchange. Since those mechanisms can vary from an SoS to another, it may be necessary to deploy different instances of a system over time. Thus, SPL could be a natural source of candidate systems to compose SoS if appropriate variability

mechanisms are applied. Those mechanisms must support late requirements because the SoS that the system is going to compose and its requirements will mostly be known just-in-time. This may require changes in conventional SPL processes based on foreseeable variabilities.

Currently, neither service orientation nor SPL evolutions seems to march to the SoS direction. Services are still strongly associated to SOA (Erl, 2008) and usual implementations of SOA solutions often create dependency relationships between consumers and providers. This is discouraged in SoS projects. SPL is mostly focused on domain specific problems whose solutions rarely require the SPL instances to be SoS members. Thus, the composition capability and its related problems are rarely addressed and the possible benefits are left aside.

In this thesis we are motivated by the challenge of contributing with possible solutions to the problems discussed above. Our proposals to the SoSE requirements, analysis and design areas are presented and discussed in the remaining chapters. Our focus, however, is the research of solutions to deploy SoS by composing conventional systems. We are aware that the results may not apply generically to SoS as a composition of large-scale complex systems. However, this is practically an unexplored area and we intend to find out alternative SoS based solutions for today's problems. Moreover, we want to go further and explore the concept of virtual SoS, which is rarely discussed and exemplified in the literature. We think it can make SoS more attractive to systems engineers and increase their interest in joining SoS based solutions.

1.3. Objectives

The objective of this thesis is bridging SE gaps towards SoS development, i.e., identifying and attempting to overcome constraints in different SE areas that currently disable or hinder conventional systems to operate regularly and still compose SoS when necessary. The expected results include an SoSE process and complementary SE approaches that fit this process properly and help us to deploy SoS in an evolutionary manner.

Guided by the this SoSE process and a sample virtual SoS that fully comprises the distinguishing SoS characteristics described by Boardman and Sauser (2006), this thesis intends to explore SoS deployment from different SE perspectives that include requirements, analysis, design, and reengineering. In Requirements Engineering (RE), we aim to research an approach able to describe SoS in an evolutionary manner, i.e., goals to be achieved, roles to be played, interactions to be performed, and collaborations to be accomplished. In analysis, we aim to research appropriate notations able to improve the modeling of interactions among

systems. In design, we aim to research how SPL can become a natural source of SoS members with reduced side effects on productivity and maintainability. Finally, in reengineering, we aim to research how to evolve legacy systems to SoS members.

Those researches will intentionally focus on the attempting to deploy a sample virtual SoS that comprises conventional systems only. SoSE approaches will be proposed and applied complementarily to deploy a virtual SoS, i.e., member systems that interact voluntarily and without coordination to achieve the SoS goal and expand their own purposes conveniently. This is a challenge because to the best of our knowledge there is no functional example of virtual SoS described in the literature. Indeed, the available examples require at least a minimal of coordination, e.g., the IOOS (2013) and the GEOSS (2013).

1.4. Organization of the thesis

This thesis is organized as follows. In Chapter 2, we present some background to ease the understanding of the remaining chapters. In Chapter 3, we propose a generic SoSE process to support SoS development and describe how the SE approaches proposed in the following chapters are used within this process. In Chapter 4, we propose a RE approach to SoS to help engineers to describe SoS and their dynamism. In Chapter 5, we propose statechart extensions to visually improve the modeling of systems' interactions. In Chapter 6, we propose extensions to the traditional SPLE process in such way that SPL can become a natural source of SoS members. In Chapter 7, we propose a reengineering approach extension to evolve legacy system to SPL and then to SoS members. Finally, in Chapter 8, we present our conclusions and future work.

Whenever a different SE area is discussed in the following chapters, a separate section of related work is presented. This allows us to make comparisons and discuss contributions specifically to each research area included in this thesis.

Chapter 2

Background

2.1. Initial considerations

In this chapter, we provide some background to enable a better understanding of further discussions involving the presented topics. Because some of these topics comprise several works that propose variations of the same main idea, only the most comprehensive and representative information are presented and discussed. Whenever possible, works that summarize and compares similar techniques and technologies are used as reference to keep this chapter short but still elucidative.

This chapter is organized as follows. In Section 2.2, we present some SoS definitions present in the literature and discuss the SoS definition we adopt in this thesis. Moreover, we present SoS classifications and distinguishing characteristics, and use them to define the scope of this thesis. Finally, we discuss traditional SE strategies and SoSE objectives to introduce the research areas covered in this thesis. In Section 2.3, we introduce SPL and discuss the challenges of integrating SPLE and SoSE. We discuss service orientation and interaction mechanisms in Section 2.4. In Section 2.5, we introduce statecharts and the visual modeling of systems' interactions. Finally, in Section 2.6, we present our final considerations.

2.2. System of Systems (SoS)

Any construct that we label a system in fact may be composed of several constituent systems, and this may be true recursively at several levels. In other words, anything that at one level we can call a system may internally be a system of systems, and any system of systems may itself be part of some larger system of {systems of systems}, and so forth (Figure 5) (Smith II, 2006).

While there is a growing recognition of the importance of SoS, there is still little agreement on just what they are or on by what principles they should be constructed (Maier, 1998). Actually, different SoS definitions and classifications can be found in the literature (Jamshidi, 2009). They are described and discussed in the following sections.

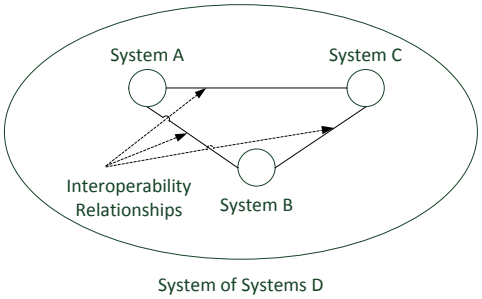


Figure 5. System and System of Systems (Smith II, 2006)

2.2.1. SoS definitions

The literature survey on SoS conducted by Jamshidi (2009) confirmed Maier's (1998) statement that the term "System-of-Systems" has no clear and accepted definition. Mostly, the definitions are influenced by the SoS application, e.g., military, education and enterprise. Three examples are given in the following.

- SoS exists when there is a presence of a majority of the following five characteristics: operational and managerial independence, geographic distribution, emergent behavior, and evolutionary development. *Primary focus*: Evolutionary acquisition of complex adaptive systems. *Application*: Military. (Sage and Cuppan, 2001).
- Enterprise SoS engineering is focused on coupling traditional systems engineering activities with enterprise activities of strategic planning and investment analysis. *Primary focus*: Information intensive systems. *Application*: Private enterprise. (Carlock and Fenton, 2001).
- SoS engineering involves the integration of systems into systems of systems that ultimately contribute to the evolution of a social infrastructure. *Primary focus*: Education of engineers to appreciate systems and interaction of systems. *Application*: Education. (Luskasik, 1998).

In this thesis, we adopt the more comprehensive SoS definition given by Boardman and Sauser (2006): "*SoS is a collection of independent entities and their interrelationships gathered together to form a whole greater than the sum of the parts. Thus, our attention is directed to parts (the entities), relationships (between the parts) and the whole itself which we understand in some sense to be better than the collection of parts and relationships*".

It is important to notice that the above definition impose nothing about the structure of the SoS members, e.g., small or large, and simple or complex, thus, it fits well our purpose of composing conventional systems. Both system and SoS terms conform to the accepted definition of system in that each consists of parts. The distinction lies in composition, i.e., the

manner in which parts and relationships are gathered together and therefore in the nature of the emergent whole.

Because we aim to deploy SoS-based solutions by composing conventional systems, some concerns often present in the literature are not addressed in this thesis, e.g., large and complex systems' interoperability issues. However, other important concerns are addressed as operational and managerial independence of the member systems, emergent behavior, and evolutionary development. This is because they generically apply to SoS.

2.2.2. SoS distinguishing characteristics

After conducting a literature survey on SoS, Boardman and Sauser (2006) identified patterns in over 40 SoS definitions. By comparing these patterns and differences against previously identified patterns of other systems, they presented a comprehensive overview of five distinguishing SoS characteristics: autonomy, belonging, connectivity, diversity and emergence. These characteristics are presented in Table 1.

Table 1. SoS distinguishing characteristics (Boardman and Sauser, 2006)

Element	System	System of Systems
Autonomy	Autonomy is ceded by parts in order to grant autonomy to the system	Autonomy is exercised by constituent systems in order to fulfill the purpose of the SoS
Belonging	Parts are akin to family members; they did not choose themselves but came from parents. Belonging of parts is in their nature.	Constituent systems choose to belong on a cost/benefits basis; also in order to cause greater fulfillment of their own purposes, and because of belief in the SoS supra purpose.
Connectivity	Prescient design, along with parts, with high connectivity hidden in elements, and minimum connectivity among major subsystems.	Dynamically supplied by constituent systems with every possibility of myriad connections between constituent systems, possibly via a net-centric architecture, to enhance SoS capability.
Diversity	Managed i.e. reduced or minimized by modular hierarchy; parts' diversity encapsulated to create a known discrete module whose nature is to project simplicity into the next level of the hierarchy	Increased diversity in SoS capability achieved by released autonomy, committed belonging, and open connectivity
Emergence	Foreseen, both good and bad behavior, and designed in or tested out as appropriate	Enhanced by deliberately not being foreseen, though its crucial importance is, and by creating an emergence capability climate, that will support early detection and elimination of bad behaviors.

Autonomy: Constituent systems must primarily be free to pursue their own purposes, i.e., they should not behave as merely parts to be able to contribute to the SoS goal.

Belonging: SoS engineers foresee the SoS partly by wondering how existing systems can play specific roles. If playing a role means changing a system, then systems engineers will have to be persuaded of the benefits – to change, to render service, and to collaborate with other systems.

Connectivity: The SoS member systems themselves have an important role to play in determining the connections they wish to establish with one another, and with new additions. To be consistent with constituent system's autonomy and its choice to belong, decisions about connectivity are taken dynamically, with interfaces and links forming and vanishing as the need arises.

Diversity: SoS should be highly diverse in its capability as a system compared to the bounded functionality of a constituent system, limited by design. A great variety of functions enables the SoS to better respond to uncertainties, surprises, and disruptive innovations.

Emergence: Emergent behaviors, resulting from the systems' autonomy and the diversity of possible connections, can increase the SoS agility to quickly detect and destroy unintended behaviors. The challenge for the SoS engineers is to create an environment in which emergence can flourish as the SoS progresses through its series of stable states.

The above SoS characteristics transform the traditional system qualities to the qualities of SoS in a paradoxical way. The paradoxes are presented in Table 2.

Table 2. Paradoxes of SoS (Sauser, Boardman, and Gorod, 2009)

Traditional system		SoS
Conformance	⟷ Autonomy	Independence
Centralization	⟷ Belonging	Decentralization
Platform-centric	⟷ Connectivity	Network-centric
Homogeneous	⟷ Diversity	Heterogeneous
Foreseen	⟷ Emergence	Indeterminable

Because the SoS distinguishing characteristics described above are general, they are adopted throughout this thesis, for example, to support the design of a sample SoS, part of our objective (see Section 1.3).

2.2.3. SoS classification

Jamshidi (2009) justifies and defines an SoS coordinator as follows: "*The addition of a new system in the SoS to cooperate in achieving a common task may change the SoS dynamics and the overall performance of the constituent systems. This event may result in emergent behaviors that are normally not addressed by systems engineers during the design process. A generic solution would be to centralize the decentralized SoS, i.e., centralize the control and information flow among the constituent systems through a central command called coordinator. The coordinator acts as a central manager to make decisions on the acceptable threshold levels for the SoS in the given application scenario*".

According to the role of the coordinator, Jamshidi (2009) classifies SoS as:

Central or Closed SoS: The coordinator has complete control over the collaboration among the constituent systems. Information and control flow among systems pass through the coordinator.

Decentralized or Open SoS: The coordinator has not complete control over the collaboration among the constituent systems. Systems can directly interact with each other, for example, through local controllers. Usually, only critical and advisory information is controlled by the coordinator.

Virtual SoS: The coordinator does not exist. Constituent systems interact freely and directly with each other, and exhibit self-configuration and self-organization to fulfill the objectives of the SoS.

Maier (1998) classifies SoS in a different way based on managerial characteristics. He defines three types of SoS:

Directed: The integrated SoS is built and managed to fulfill specific purposes. It is centrally managed during long-term operation to continue to fulfill those purposes as well as any new ones the system owners might wish to address. The constituent systems maintain an ability to operate independently, but their normal operational mode is subordinated to the central managed purpose.

Collaborative: The constituent systems interact more or less voluntarily to fulfill agreed upon central purposes. Central players collectively decide how to provide or deny services, thereby providing some means of enforcing and maintaining standards.

Virtual: They lack a central management authority and a centrally agreed upon purpose for the SoS. Large-scale behavior emerges, but this type of SoS must rely upon relatively invisible mechanisms to maintain it.

From the given definitions it is possible to formulate several questions about SoS. For example: How to make a consistent and secure integration of new systems? How to detect and effectively correct deviations of behaviors? How to dynamically evolve the SoS goal? How are changes in the constituent systems propagated throughout the SoS? How interdisciplinary systems must perform collaborative tasks?

Answering these questions may not be a simple task if complex SoS are considered. The sample SoS used in this thesis is comprised of a small number of conventional systems. The objective is to be able to develop it systematically, to exemplify the SoS distinguishing characteristics of Table 1, to discuss their *pros* and *cons*, and to evaluate our proposals to extend traditional SE approaches towards SoS development. Thus, the questions above may not be totally answered.

Particularly in this thesis, the sample SoS we aim to deploy is a virtual SoS according to Jamshidi (2009), i.e., a collaboration among systems without a coordinator. Constituent systems are free to pursue their own purposes and still voluntarily contribute to the SoS goal. This type of SoS is particularly rare to find in the literature, thus we think we can improve the contribution of this thesis by describing, designing and deploying a virtual SoS with the support of new or extended SE approaches.

2.2.4. Family of Systems (FoS)

Family of Systems (FoS) is fundamentally different from SoS. It is a set of different systems belonging to a common domain, possibly designed with different approaches, which are able to behave similarly (Clark, 2009). An example is a set of cameras from different manufacturers as shown in Figure 6. FoS does not acquire qualitatively new properties as a result of the grouping. In fact, the member systems may not even be connected into a whole like the given example.

Figure 6. FoS of cameras

FoS can play an important role in the SoS context as a supplier of constituent systems. Initially, engineers entirely specify the SoS considering just the abilities necessary to achieve the SoS goal. At this time, constituent systems are just black boxes and do not need to be assigned to specific systems. When finished, different FoS can be searched for qualified systems to replace the black boxes. For example, if capturing video is an important ability to

the SoS goal, the FoS of Figure 6 can provide to SoS engineers a qualified list of candidate systems to support decision making. Figure 7 illustrates a scenario where different SoS are supported by the FoS of Figure 6.

Figure 7. Different SoS supported by the FoS of cameras

In the above example, we are considering that the constituent systems, e.g., cameras and vehicles are conventional systems that can be built on demand, operate independently and interact with each other. This way, the SoS can be designed more systematically.

2.2.5. SoS engineering (SoSE)

According to Jamshidi (2009), the major issue facing SoS and SoSE is that traditional SE needs to undergo a number of innovative changes to accommodate and encompass SoS. To him, SoSE challenges lies in extending traditional SE concepts such as analysis, control, estimation, design, modeling, controllability, observability, stability, filtering, simulation etc. so they can be applied to SoS. However, it is not consensual that traditional SE and SoSE are different. Clark (2009), for example, believes that the traditional SE processes as documented in the SE standards and guides: IEEE 1220, EIA/IS-632, EIA-632, ISO 15288, and ISO TR 19760, are a necessary and sufficient set of processes for SoSE, and no additional processes are needed. To Azani (2009), in the 21st century, systems engineers must deliver systems with the ability to be self-organized and self-regulated, and be reconfigured affordably and very quickly in response to evolving needs and rapidly changing technologies. According to him, traditional SE strategies do not comply with those needs.

Keating et al. (2003) made several observations relevant to engineering an SoS and discussed why traditional SE practices are often not sufficient for that purpose. They noted several important realities concerning an SoS. They are summarized in Table 3.

To Azani and Khorramshahgol (2005) many of the premises underlying the traditional SE strategies are no longer valid for SoSE. Traditional SE practices have been focusing on developing stand-alone systems with stable architecture and static technology that cause improvements to be slow and very costly. These strategies incorrectly assume that all the SoS requirements are known in the beginning of the development process and can be frozen in

time or assumed to be stable. According to Azani (2009), traditional SE strategies wrongly assume that the concepts of operation and various technologies used for constructing today's SoS are static and are subject to minor future changes.

Table 3. Traditional x SoS engineering (Keating et al., 2003)

Critical Point	System Engineering	SoS Engineering
Focus of Analysis	Single System	Integration of Systems
Focus of Improvement	Optimization	Realistic Cost and Scheduling
Target	End Product	Initial Deployment
System Requirements	Fixed	Evolving
System Boundaries	Well-defined	Indefinable

Because Clark (2009) refers to SE processes and not strategies or practices to handle SoSE problems, he states that SE and SoSE are similar. However, Table 3 shows why they are not by comparing some critical points. In this thesis, we consider SE and SoSE are not similar and conventional SE approaches cannot handle SoSE problems without extensions or new proposals. In the following chapters we will discuss this statement for different SE areas.

2.2.6. SoS composition

Systems' integration may occur distinctly in different SoS as follows:

Indirect integration: The integration of complex systems may be a hard task even out of the SoS scope. It can be done indirectly integrating information rather than systems (Morris et al., 2004). The term "information intensive system" is popular in the SoS literature and usually refers to complex systems (vertical) responsible for deploying high quality information to other SoS systems (transversal). This enables vertical systems to operate independently and evolve freely. An example is the SoS of Figure 8 (IOOS, 2013) that provides regularly high quality data and information about the current and the future state of the oceans and the great lakes on a global scale (ocean basins) and local scale (coastal eco systems). Vertical systems acquire data and deploy information to transversal systems, which are engineered to mine, organize, merge and format this information and deliver new information-based services to plenty of users, institutions and companies.

Although information intensive systems like the IOOS are so called SoS, they may not fully comply with the SoS distinguishing characteristics of Table 1. Actually, there may be

direct dependencies between transversal and vertical systems that cause the first ones to not have their "own lives" out of the SoS.

Figure 8. The U.S. Integrated Ocean Observing System (IOOS, 2013)

Direct integration: SoS constituent systems interact directly with each other by means of proper interaction mechanisms like service orientation (Lewis et al., 2011). Direct systems' integration usually requires either the reengineering of legacy systems or the development of new systems to meet the SoS requirements. While this is an option to integrate smaller and less complex networked systems it is also a source of misunderstandings. Indeed, designing new SoS members does not mean building high specialized systems dedicated exclusively to the achievement of the SoS goal. In such case, these systems would be merely parts of a whole.

The service-oriented SoSE approach proposed by Lewis et al. (2011) is illustrated in Figure 9. SoS end users present new capability requirements. SoS developers/integrators analyze the requirements, search the capabilities repository for matching capabilities and analyze the search results. If there is not a match, SoS developers/integrators can either find other alternatives or request new development. Otherwise, if the requirements and capabilities do not fully match, they adapt the existing capability to deal with the mismatches. Then, they integrate the new capability into the SoS, test and validate it and its effect on the SoS, and deploy the new capability to SoS end users.

In this thesis, the sample virtual SoS we aim to deploy requires the direct integration among the constituent systems. Particularly, the integration is service-oriented. The goal is using services to create platform-independent interaction mechanisms that will not suggest dependencies among member systems. Our proposal is different from the one of Figure 9 which seems to be a central SoS with the SoS itself playing the role of coordinator.

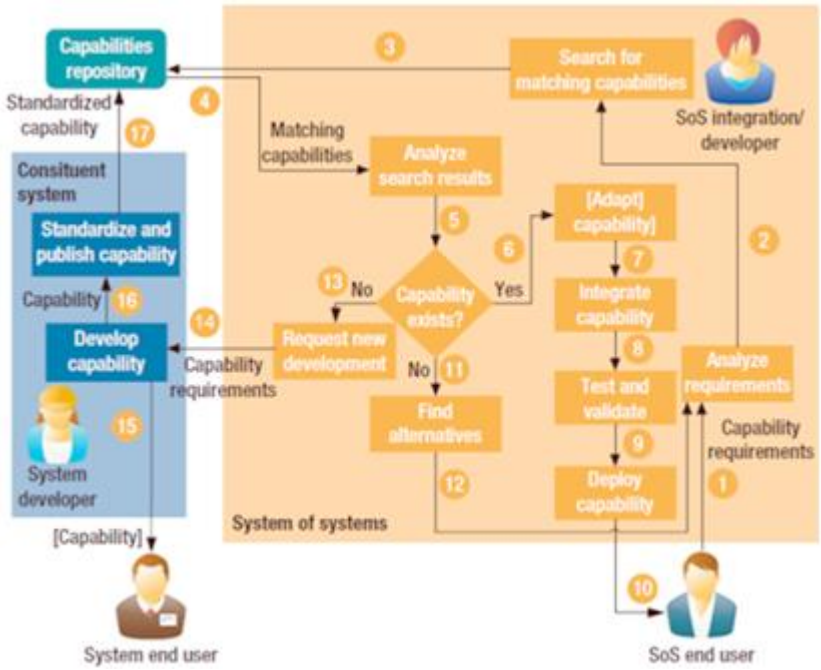


Figure 9. Service-oriented SoSE approach (Lewis et al., 2011)

To perform a direct integration among systems, SoS engineers should not focus on describing systems and their particularities as usual in conventional SE practices. Instead, the focus should be directed to the description of roles, interaction mechanisms and collaboration rules. This fact points to a different strategy for eliciting SoS requirements and possibly to the necessity of new approaches to support it. Because of this necessity, we propose, in Chapter 4, a requirements engineering approach for eliciting SoS requirements, suitable to our purpose of composing conventional systems to deploy a sample virtual SoS.

2.2.7. SoS modeling

Zhou et al (2011) have performed a review of the existing methods for modeling SoS. The SoS they used as basis complies with the SoS distinguishing characteristics discussed previously in Section 2.2.2 and shown in Table 1. Three groups of methods were reviewed:

Traditional System Methods: The SoS is decomposed until every fragment is an individual traditional system, and then traditional systems engineering principles are applied. There is no need for developing new modeling principles;

SoS-Specific Methods: The SoS cannot be treated just as a simple congregation of several systems to be decomposed into fragments on a certain basis. Thus, traditional engineering methods are seen inadequate as to design SoS. The most important design problem expected to be solved in SoSE is related to how observed behaviors of an SoS affect its members including their connectivity and autonomy. Important is that physical manifestations may be

most obvious at the member level (α-level), but the behavior of the SoS is dominated by the structure and organization at the SoS level (β-level). The SoS dynamism is shown in Figure 10.

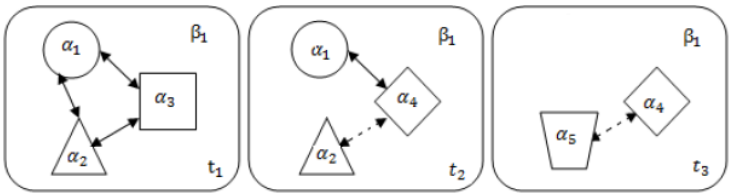


Figure 10. SoS dynamism (Zhou et al. 2011)

In a certain moment t1, SoS members α1, α2 and α3 interact with each other. In a moment t2, α4 replaces α3 that leaves. α4 interacts differently with α2. Finally, in a moment t3, α5 replaces α2. α1 and α2 leave. At each moment, the physical manifestations of the members can be observed individually at the α-level while the related SoS behavior can be observed at the β-level.

Generic method based on modeling characteristics and interface: This method combines together the previous approaches for modeling the SoS characteristics (properties of each component) and the SoS interface (communication among components). It describes member systems together with their goals and functionality and provides a coherent model of the SoS.

In this thesis, we propose, in Chapter 5, an SoS modeling approach that complies with the last method described above. It first describes the SoS in terms of communicating components to elicit roles, interactions and collaborations necessary to achieve the SoS goal. Then, each component is described in terms of behavioral interfaces. The goal is to enable SoS engineers to design the SoS without being aware of existing systems. This approach is different from those present in the literature that consider the existence of candidate systems and useful abilities to design the SoS. Our proposal is particularly useful for composing conventional systems that, different from large and complex systems, can be built or maintained on demand more easily to get benefits of belonging to the SoS.

2.3. Software Product Line (SPL)

Software Product Line (SPL) was introduced in the 80's to improve productivity through software reuse (Clements and Northrop, 2001). Krueger (2006) discussed the current generation of intentional SPL initiatives, where organizations adopt with forethought the best practices discovered from the practical experiences of their predecessors. He describes new methods that have provided some of the most significant advances to SPL practice. However,

the original proposal of designing domain-specific families of related end products has kept unchanged and the resulting systems are mostly not designed to pursue purposes that require collaboration with other systems. Thus, there is a gap between SPLE and SoSE that must be overcome before they can contribute mutually.

2.3.1. Definitions

According to Clements and Northrop (2001): "*A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*". This definition describes different roles in a product line organization as illustrated in Figure 11. The core asset developers provide to the product developers the resources necessary to produce the target products. This includes the architecture, the system components that populate the architecture, plans such as production plans and test plans, and templates for process definitions. At the variation points of the SPL, multiple assets are designed and implemented to cover the possible product permutations. Product line managers coordinate and facilitate the work of both developer teams.

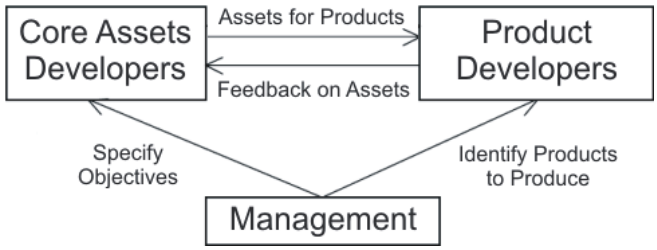


Figure 11. Roles in a product line organization (McGregor, 2004)

2.3.2. General SPL Engineering (SPLE) process

The general process for SPLE described by Ziadi, Jézéquel, and Fondement (2003) and illustrated in Figure 12**.** It is constituted of Domain and Application engineering phases.

Domain Engineering includes the activities of Analysis, Design and Implementation of the domain that, respectively, elicits the SPL requirements, designs the SPL architecture and develops reusable software components. The resulting artifacts constitute the core assets of the SPL. Application Engineering includes the activities of Analysis, Design and Implementation of products that meet the customers' needs. Whenever those needs are not fully supported by the current core assets, feedback is given to the domain engineers who must perform a feasibility study to improve them. Then, the requested products can be deployed.

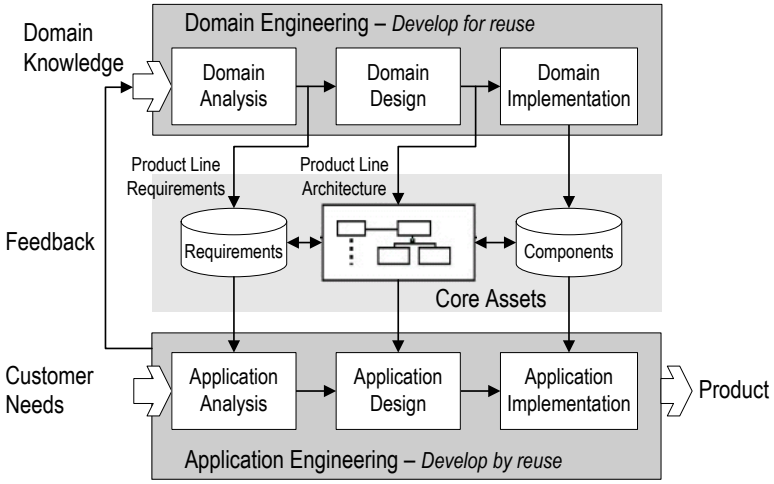


Figure 12. General process for SPLE (Ziadi, Jézéquel, and Fondement, 2003)

A major SPLE challenge can be deduced from Figure 13**.** The productivity (high reuse rate) and profitability of the SPL strongly depend on the completeness and quality of its core assets. The more feedback is given (low reuse rate) the greater are the effort and investment necessary to maintain the product line. This moves continuously the 'Break even' point forth in the graphic. The result is the decrement of the SPL overall productivity and profitability.

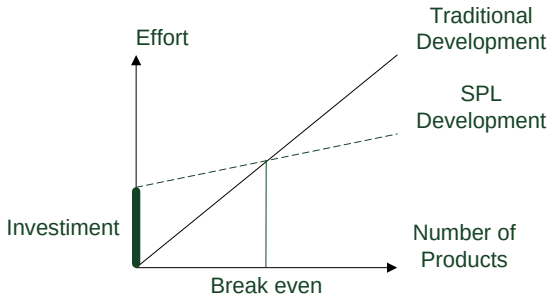


Figure 13. SPL effectiveness (Clements and Northrop, 2001)

The design of SoS constituent systems would mean extra challenges for SPLE and could move further the 'Break even' point. These challenges include shared abilities and interactions.

Shared abilities: The achievement of the SoS goal often requires the sharing of abilities among the SoS constituent systems. SPL products, as member systems, can benefit from this to expand opportunistically their set of native abilities by aggregating new abilities shared by other SoS members during the SoS operation. Although terms like dynamic features, features as services (Lee and Kotonya, 2010) and context-aware features (Cetina, Fons, and Pelechano, 2008) have been already proposed in the literature, they refer mostly to the native features of a particular SPL and their activation/deactivation mechanisms which are defined

usually at design time. On the other hand, mechanisms to aggregate shared abilities may vary from one SoS to another depending on the SoS specific characteristics. Consequently, these mechanisms can hardly be specified at design time and may require increasing efforts to maintain the core assets.

Interactions: SoS constituent systems are intended to run independently but still contribute collectively to the SoS goal. Different behaviors may be necessary from one SoS to another because of the different goals they may have and the different interactions they may require. Similarly to shared abilities, interactions can hardly be specified at design time and the same problem applies.

The challenges discussed above highlight some of the reasons why conventional SPLE approaches commonly are not a natural choice for deploying SoS members.

2.3.3. Dynamic SPL (DSPL)

According to Capilla et al. (2014), today's SPL models are limited by their inability to change the structural variability at runtime, provide dynamic selection of variants, or handle the activation/deactivation of features dynamically and/or autonomously. They argue that the development of runtime reconfigurable assets is still innovative and not fully investigated in the SPL area. Thus, there is an important need to support dynamic properties of systems and post-deployment capabilities. Dynamic SPL (DSPL) is an emerging research area focused on bridging such gap.

Efforts that suggest the use of DSPL-based models focus on the implementation of runtime variability mechanisms and domain-specific languages for reconfiguring software systems. Two examples are given in the following.

The SPL model of Figure 14 shows foreseen features of a home robot able, for example, to attend to a voice call ("*Call and Come*") through more specialized actions like localization (*"Caller Localizing"*) and movement ("*Go Forward and Rotate"*).

Features can be activated or deactivated at runtime according to the context. For these transitions to occur properly it is necessary to foresee the influence of each feature over the other ones and also over the set of structural and behavioral elements shared between them, With such knowledge, it is possible, for example, to activate a particular feature dynamically while preserving the stability of the system by temporarily disabling and reconfiguring other features influenced negatively by the feature becoming active. The features of the home robot of Figure 14 are managed that way.

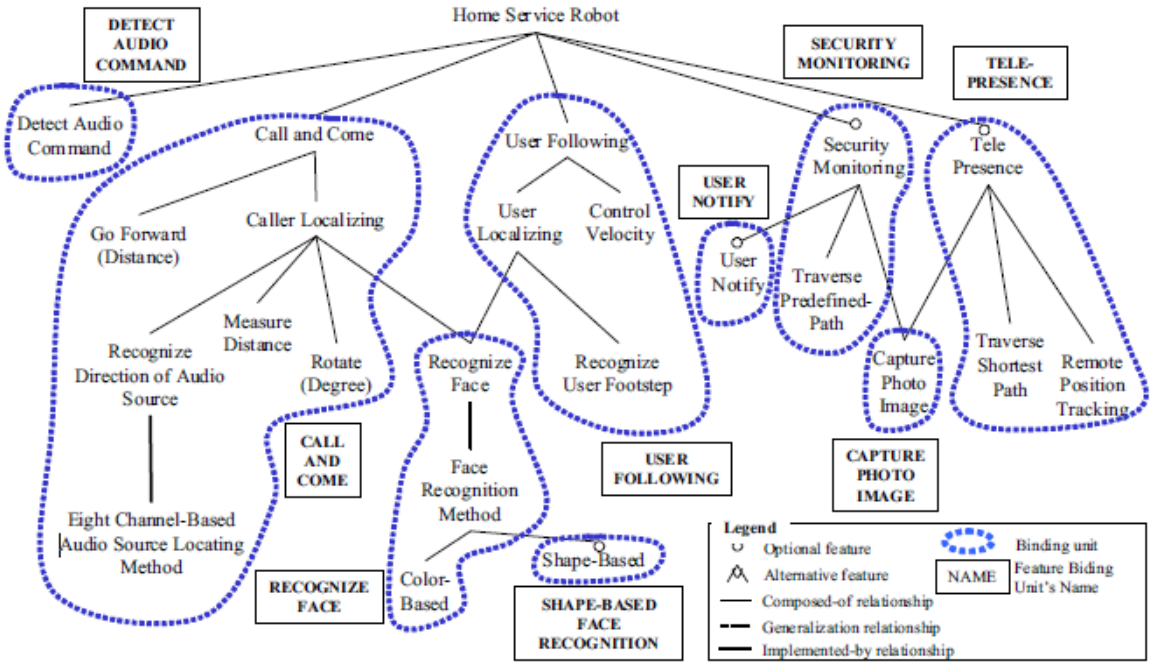


Figure 14. Home robot feature model (Lee and Kang, 2006)

Cetina et al. (2009) proposed a context-aware DSPL for smart homes. Figure 15 shows different configurations of the DSPL model for two distinct scenarios: a) user at home and b) user out of home.

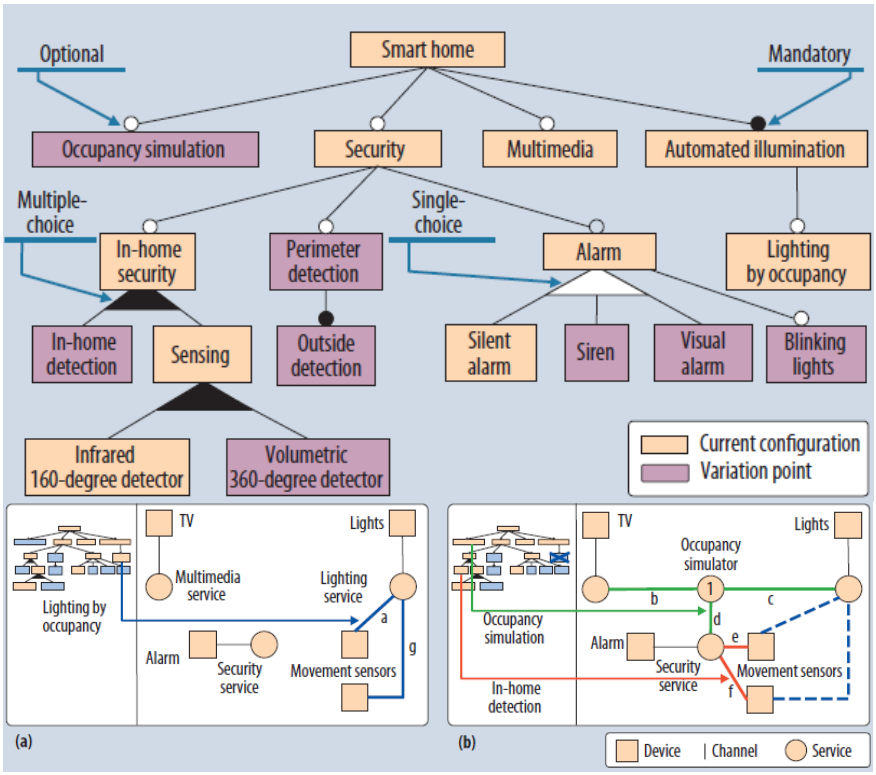


Figure 15. Distinct configurations of the context-aware DSPL for smart homes: a) user at home e b) user out of home (Cetina et al, 2009)

Common structural elements, e.g., lights, can play different roles depending on the active scenario. For example, when the user is at home (a), the lights automatically illuminate the user path as he/she walks throughout the house ("*Lightning service*"). When the user leaves the house (b), the lights can simulate the user presence as well as indicate an alarm condition ("*Occupancy simulator*").

These two examples of DSPL highlight the gap that exists between SPLE and SoSE as we introduced in Section 2.3. Indeed, these systems aim to solve domain-specific problems standalone as usual, thus collaboration capabilities were not even considered in both projects, for example, to improve their own purposes. When both systems are put to operate in the same environment the consequences become more evident. The robot walking in the house, for example, can trigger movement sensors and activate the scenario (a) of Figure 15 even when the user is out of home. In this scenario, simulation occupation is disabled as well as some security services. On the other hand, if both systems could interact, such situation could be avoided and more advanced features could be delivered by means of collaborations.

It is true that interoperability can be delivered as dynamic features in DSPL projects. However, the solution is not just a matter of setting them on or off like in the given examples. Indeed, it would be necessary to redesign those features for every different interaction and collaboration rules that were not foreseen at design time. This is a common scenario in the SoS context where those rules can change even after the SoS has been deployed because of its evolutionary nature.

As discussed above, although DSPL projects can handle dynamic features, they can mostly be foreseen at design time like in the given examples. This way, handling late requirements, like interaction and collaboration rules, is still a challenge to integrate SPLE and SoSE and will be addressed in this thesis, Chapter 6, as part of the objective to deploy a sample virtual SoS.

2.3.4. SPLE and SoSE integration challenges

SPLE can support the deployment of SoS by designing families of systems more likely to participate of compositions. However, as previously discussed, SPLE aims to deploy domain-specific solutions mostly not intended to collaborate with each other, for example, the approaches proposed by Clements and Northrop, 2001; Atkinson et al., 2001; and Gomaa, 2004. Thus, SPLE would need to expand its boundaries to match SoSE needs. This expansion includes, but is not limited to, the following challenges:

- Sharing abilities may require more opportunistic SPL approaches;

- Collective behaviors may require the deployment of collaborative systems;
- Evolunionary SoS requirements may require improved SPL reference architectures;
- SoS uncertainties may require a natural handling of unintended behaviors.

If a family of systems is intended to compose an SoS, then both SoSE and SPLE processes may eventually overlap. For example, SoS engineers may want to know which systems they can count with to compose the SoS and suggest their roles and collaboration interfaces. On the other hand, SPL engineers may want to know how their systems can benefit from belonging to the SoS. Thus, SPLE may require extra information from SoSE to fully elicit the SPL requirements other than those provided by domain specialists and stakeholders. Similarly, SoSE may require extra information from SPLE to complete the SoS requirements.

SPLE approaches that intend to build SoS constituent systems have to deploy products compliant with the SoS distinguishing characteristics of Table 1, i.e., autonomy, belonging, connectivity, diversity and emergence. This is a challenge since the concepts that establish the SPL boundaries should evolve compulsorily, for example, to achieve supra-purposes (the SoS goal) collaboratively. Systems' integration must occur in such way that each member can still preserve its identity and individuality. SPLE can deal with systems' integration but may have difficulties to adapt quickly its systems to different SoS solutions. A possible solution for this problem is proposed in Chapter 6, as part of the objective to deploy a sample virtual SoS.

2.4. Service orientation

In this thesis, service-orientation plays the important role of supporting platform-independent interaction mechanisms. However, an important distinction should be made between service-orientation and Service-Oriented Architecture (SOA).

According to Erl (2008) "service-oriented" is an ideal vision of the world in which resources are clearly partitioned and consistently represented. The term represents a distinct approach for separating concerns. It means that logic required to solve a large problem can be better constructed, carried out, and managed if it is decomposed into a collection of smaller, related pieces. Each of these pieces addresses a concern or a specific part of the problem. This approach transcends technology and automation solutions.

When coupled with "architecture", the resulting term establishes a universal model in which automation logic and even business logic conform to this vision. Thus, "Service-Oriented Architecture" (SOA) is a term that represents a model in which automation logic is

decomposed into smaller, distinct units of logic. Collectively, these units comprise a larger piece of business automation logic. Individually, these units can be distributed.

It is important to notice that SoS as a composition of systems with different abilities that act collectively to achieve a common goal conforms to the service-oriented vision. Thus, service-oriented SoS transcends technology and must not be wrongly associated to SOA.

2.4.1. The WS-* standards

One of the SoSE focus is defining appropriate interoperability mechanisms to connect heterogeneous systems (Morris et al., 2004). In this thesis, we particularly adopt the WS-* standards to support interactions among SoS constituent systems.

The term WS-* has become a commonly used abbreviation that refers to the second-generation Web services specifications. These are extensions to the basic Web services framework established by first-generation standards represented by WSDL, SOAP, and UDDI (See Erl, 2008 and Erl et al, 2008 for details).

Specifically to our purpose, WS-* standards provide loosely coupling interaction mechanisms with improved composability, interoperability, reusability, extensibility, and discoverability. Next, we summarize the most important WS-* standards for the context of this thesis.

WS-Addressing standard provides transport-neutral mechanisms to address Web services and messages (WS-Addressing**,** 2004). It defines XML elements to identify Web service endpoints and to secure end-to-end endpoint identification in messages. The content of the message information header is described in the following.

<wsa:MessageID>	xs:anyURI	</wsa:MessageID>	Unique ID of the message
<wsa:RelatesTo>	xs:anyURI	</wsa:RelatesTo>	Original message ID for replies to this message
<wsa:To>	xs:anyURI	</wsa:To>	Intended receiver of this message
<wsa:Action>	xs:anyURI	</wsa:Action>	Action to be performed
<wsa:From>	*endp-reference*	</wsa:From>	Where the message originated from
<wsa:ReplyTo>	*endp-reference*	</wsa:ReplyTo>	Intended receiver for replies to this message
<wsa:FaultTo>	*endp-reference*	</wsa:FaultTo>	Intended receiver for faults

WS-Discovery standard defines a discovery protocol to locate services (WS-Discovery**,** 2009). It enables the discovery of services in ad hoc networks with a minimum of networking services (e.g., no DNS or directory services). To find a target service by type in ad hoc mode, a client sends a *Probe* message to a multicast group; target services that match the probe send a response (*Probe Match*) directly to the client. To minimize the need for polling, when a target service joins the network, it sends an announcement message (*Hello*) to the same multicast group. By listening to this multicast group, clients can detect newly available

target services without repeated probing. The same applies when a target service is going to become unavailable. It sends a best-effort message (*Bye*) to the same multicast group when it leaves a network. By listening to this multicast group, clients can detect target services becoming unavailable without repeated probing. Figure 16 illustrates such protocol.

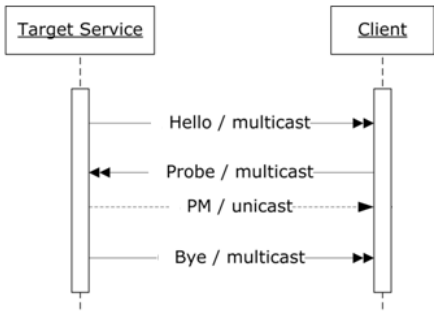


Figure 16. WS-Discovery protocol (ad hoc mode) (WS-Discovery, 2009)

In a managed mode, discovery messages are sent unicast to a Discovery Proxy. A Target Service sends a unicast *Hello* message to a Discovery Proxy when it joins a network. A Client sends a unicast *Probe* request to a Discovery Proxy to locate services. A Discovery Proxy responds to a unicast *Probe* request with a *Probe Match* response containing matching Target Services, if any. A Target Service makes an effort to send a unicast *Bye* message to a Discovery Proxy when it leaves a network. Figure 17 illustrates such protocol. To limit multicast traffic, Clients may dynamically switch from an ad hoc mode to a managed mode and vice versa,

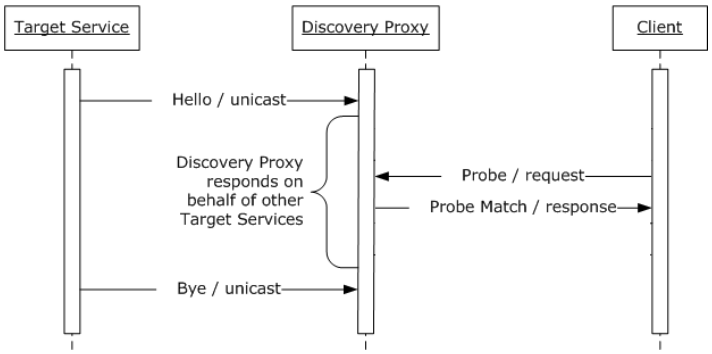


Figure 17. WS-Discovery protocol (managed mode) (WS-Discovery, 2009)

WS-Notification standard defines a set of specifications that standardize the way Web services interact using "Notifications" or "Events" (WS-Notification**,** 2006). They can be thought of as defining "Publish/Subscribe for Web services". The messages Notification, Subscribe, Unsubscribe, Pause Subscription, Resume Subscription, and Renew are defined to support publish/ subscribe interaction. It forms the foundation for Event Driven Architectures built using Web services.

2.5. Statecharts

In this thesis, statecharts (Harel, 1987) are the basis of our proposal to model systems' interactions. Thus, a short introduction to its elements must be given.

Hypergraphs (Berge, 1973) are graphs in which the relation being specified does not need to be of fixed "arity". Formally, an edge no longer connects a pair of nodes, but rather a subset thereof thus being called a hyperedge. A graphical representation of a hypergraph with unlabeled directed hyperedges of variable arity is shown in Figure 18a.

While graphs and hypergraphs are a good way of representing a set of elements together with some special relation(s) on them, Euler/Venn diagrams (Figure 18b) are a good way of representing a collection of sets, together with some structural (i.e., set-theoretical) relationships between them (subset of, disjoint from, and nonempty intersection with) (Harel, 1988).

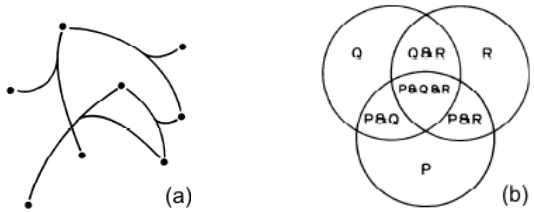


Figure 18. Hypergraph and Euler/Venn diagram

Harel (1987; 1988) observed that in numerous computer-related applications both capabilities are needed. Higraphs result from his effort to join these capabilities by first modifying Euler/Venn diagrams somewhat, then extending them to represent the Cartesian product, and finally, connecting the resulting curves by edges or hyperedges. Identifying the Cartesian product of some of the sets is crucial to prevent certain kinds of representations from growing exponentially in size.

Rounded rectangles are used to represent Euler's circles and the areas, or zones, they enclose are called blobs. Every set of interest is represented by a unique blob, complete with its own full contour. One of the reasons for this is to provide every set with its own area, e.g., for naming or labeling purposes. The other reason is to enable blobs to be connected to others via the edges.

A higraph is obtained by simply allowing edges, or more generally, hyperedges, to be attached to the contour of any blobs. As in graphs, edges can be directed or undirected, labeled or unlabeled, of one type or of several. Figure 19 shows an example of higraph (full

details about it can be found in Harel, 1998). Blob J is an example of Cartesian product and represents symbolically the notation:

$$\mathbf{J = W \otimes X = (K \cap N) \otimes (I \cap L \cap M)}$$

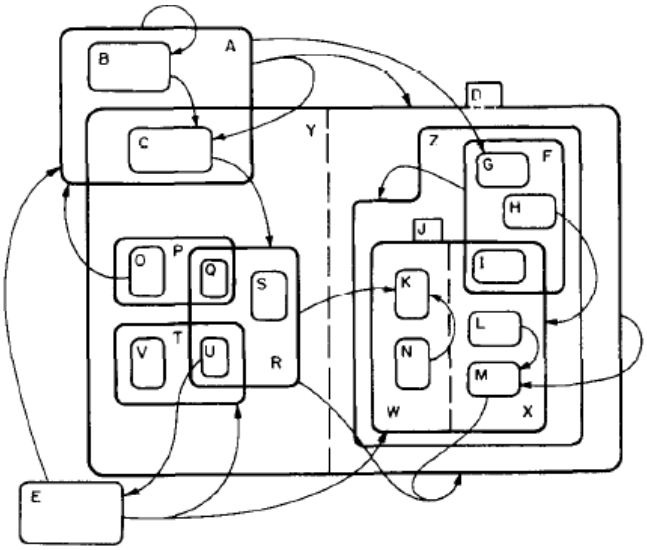


Figure 19. Higraph with Cartesian product (Harel, 1998)

One thing to notice when attempting to apply higraphs is that edges connect sets to sets, not elements to elements as in graphs. Thus, a higraph's edge can be interpreted as a collection of regular edges, connecting each element in one set with each element in the other. For example, a 5-clique behavior represented in Figure 20a is similar to the higraph of Figure 20b.

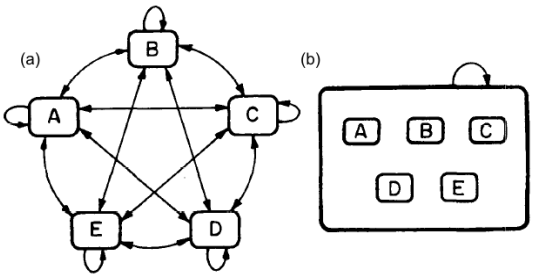


Figure 20. Similar representations of a 5-clique (Harel, 1998)

Statecharts are a higraph-based extension of standard state-transition diagrams, where the blobs represent states and edges represent transitions. They can be viewed as an extension of Mealy machines (Hopcroft and Ullman, 1979). Thus, output events, i.e., actions, can be attached optionally to the triggering event along a transition. However, in contrast with conventional Mealy machines, an action appearing along a transition in a statechart is not merely sent to the "outside world" as an output. Rather, it can affect the behavior of orthogonal components of the same statechart. This is achieved by a simple broadcast

mechanism. In the statechart of Figure 21, when the event '*a*' triggers the transition (X,W) → (Y,W) the broadcast of action '*b*' causes the transition (Y,W) → (Y,Z).

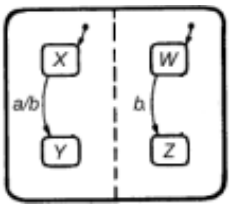


Figure 21. Statechar's broadcast mechanism (Harel, 1987)

As to the basics, we might say that statecharts extend conventional state diagrams with the notions of hierarchy, concurrency and communication:

statecharts = state diagrams + hierarchy (depth) +
concurrency (orthogonality) + communication (broadcast)

Finally, statecharts may contain overlapping states (Harel, 1987). This is the situation where in a hierarchy a state has multiple parents. The reason for doing this might be conceptual similarities between the involved states. Figure 22 illustrates the overlapping states C e D and their common parents A1 and A2 (for detailed discussions see Harel, 1987).

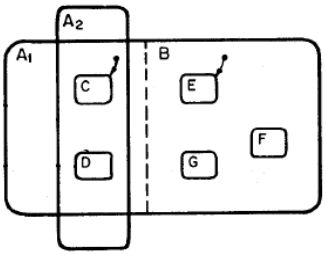


Figure 22. Ovelapping states (Harel, 1987)

Harel (1988) has proposed the use of statecharts primarily to improve the specification and design of reactive systems, i.e., those that have to react to external and internal stimuli continuously. Visual elements support the description of reactive behaviors in clear and realistic ways. However, they lack elements to describe clearly interactions among different systems, which might be fundamental to model compositions like SoS.

2.6. Final considerations

In this chapter we summarized important concepts to increase the understanding of the discussions performed in the next chapters. Moreover, we discussed constraints of traditional SE approaches towards SoS development. The attempt to overcome these constraints with feasible solutions is the basis of our proposals presented in the next chapters.

Chapter 3

A generic SoSE process

3.1. Initial considerations

As stated in Section 2.2.1, in this thesis we adopt a more general definition of SoS given by Boardman and Sauser (2006), which focuses mainly on parts, their relationships, and the improved whole. Unlike those works in the literature that focus on composing large-scale and often complex legacy systems, e.g., Huynh and Osmundson (2006), we focus on the composition of conventional systems to systematically deploy SoS. We aim to investigate and possibly confirm that in some circumstances there are advantages in deploying conventional systems as SoS. For that purpose, we need a guiding process that properly organizes the SoSE activities and focus on the critical points defined in Table 3, i.e., integration of systems, evolving requirements, and extendable boundaries. By following this process, we intend to be able to deploy SoS in a systematic and evolutionary way.

This chapter is organized as follows. In Section 3.2, we propose a guiding process for SoSE activities. Moreover, we discuss why a conventional SE process was not used instead. In Section 3.3, we discuss the activities of the proposed SoSE process that are addressed in this thesis and were fulfilled to deploy a sample virtual SoS (see Section 1.3). In Section 3.4, we describe a scenario in which the proposed SoSE process could be applied and discuss the expected benefits compared to applying conventional SE processes. In Section 3.5, we discuss related works and, finally, in Section 3.6, we present our final considerations.

3.2. The proposed SoSE process

The guiding process for SoSE we propose considers the characteristics of the systems we intend to compose, i.e., conventional systems. When those systems are not complex, they probably can be developed or maintained in a reasonable time by systems engineers to meet the SoS requirements. It means that SoS engineers need not to be fully engaged on composing legacy systems as usual although this is still the preferred approach. Indeed, they are allowed to foresee complete new systems by describing the roles that those systems must play and the interaction and collaboration rules they must comply with. This new capability increases the

diversity in SoSE. This is because several suppliers may be apt to deploy variants of each candidate system, e.g., a camera. When this happens, the set of resulting similar systems forms a Family of Systems (FoS) (see Section 2.2.4).

When FoS is considered in SoSE, it can be seen as a repository of candidate systems for a certain role. Those repositories can be systematically matched against SoS requirements and the compliant systems invited to become SoS members. These systems can still decide to belong or not to the SoS based on cost-benefit analysis.

The SoSE process we propose to support the deployment of a virtual SoS is shown in Figure 23. The phases addressed in this thesis are distinctly colored as well as the flow of activities and artifacts. It is important to notice that usual SE phases like implementation are absent. This is because we are motivated by the possibility of deploying a virtual SoS, i.e., a composition of independent systems without a coordinator (see Section 2.2.3), without glue code. For us, it would be enough to deploy a virtual SoS if the constituent systems could play the required roles and prevent disruptive behaviors by directly and freely interacting with each other. In this case, SoS engineers have the critical mission of clearly and fully describing the roles, collaboration rules and interaction mechanisms necessary to make the SoS operational.

To the best of our knowledge the literature lacks proposals for virtual SoS the way we described. Indeed, typical examples of SoS like the GEOSS (Figure 1) and the IOOS (Figure 8) require some kind of coordination and glue code to make the SoS operational, i.e., to mine, organize, merge, and format information and deliver information-based services to SoS users. In both examples, SoS members are complex systems, as usual, that provide high quality data and information but do not necessarily directly and freely interact with each other.

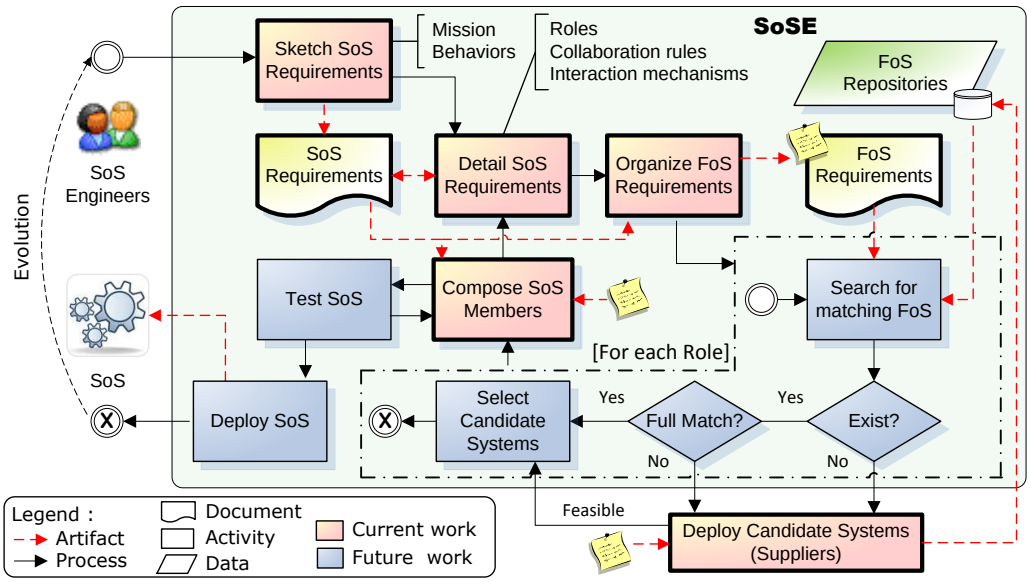


Figure 23. The proposed SoSE process

According to the SoSE process of Figure 23, SoS engineers aim to compose systems rather than to develop them. Developing systems is a role assigned to Suppliers, i.e., teams of systems engineers. The proposed SoSE process does not match SE processes commonly used to develop conventional systems. Indeed, requirements, analysis, design, implementation, and testing phases are not all defined or clearly distinguished. Even those phases that eventually match may comprise different activities and goals. For example, the requirements engineering (RE) activities, i.e., *Sketch*, *Detail* and *Organize*, do not aim to fully describe a system and its structural and behavioral characteristics. Instead, they aim to describe the SoS goal and how the constituent systems should collaborate by means of interactions to achieve this goal. Thus, SoS engineers design interactions rather than elicit systems' information.

After the SoS has been fully described, i.e., roles, collaboration rules and interaction mechanisms, each role can be assigned to one or more candidates systems in the sub-process bordered by a dash-dotted line, which is performed once per role. In this sub-process, the requirements of each particular role are matched against a FoS repository, which contains information about systems able to become SoS members. Each FoS repository may comprise several different systems with similar functionality. Systems that meet the requirements are invited to compose the SoS. If no candidate systems can be directly assigned to a certain role, then other systems can be either developed or maintained (e.g., reengineered) by suppliers. Later, those systems will populate a FoS repository.

Composing systems in a virtual SoS is not like composing components in a system. This is because SoS members are independent systems that do not maintain part-whole relationships or may even not maintain the same behavior over time. This way, several critical situations should be taken into account, for example, systems deliberately entering and leaving the SoS in an uncoordinated way. These situations may never happen in component-based development (Heineman and Councill, 2001). Even more modern SE approaches as dynamic reconfiguration in self-adaptive systems (Baek, Han, and Chung, 2013) address dynamism in a different way. Indeed, those systems require a minimal coordination, e.g., adaptation plans, and the dynamism refers to when, how and what parts, not systems, will be (re)configured, (de)activated or replaced in different contexts at runtime. Unlike virtual SoS members, parts in self-adaptive systems are mostly not free to take decisions by themselves.

Finally, testing a virtual SoS will probably not include conventional test activities for systems, e.g., code validation. Instead, analysis and simulation of systems' interactions in dynamic scenarios must be strongly exercised.

SoSE testing activities are not addressed in this thesis as well as the matching of FoS requirements. The second would require FoS repositories, which will be available only after the SoSE process has been performed a couple of times. These activities deserve specific research and each will be addressed in future work.

3.3. The current work

In the SoSE process of Figure 23, the RE phase is carried out by three activities, i.e., *Sketch*, *Detail* and *Organize* SoS requirements. In the first activity (*Sketch*), SoS engineers strive to clearly understand the goal to be achieved. The foreseen actions include:

- Outline the SoS mission (goal, expected behavior);
- Divide the mission among players (black boxes with distinct roles);
- Decide on the rules (composition, management etc.);
- Define constraints (boundaries, regulatory laws etc.);
- Portray the SoS environment (climatic conditions etc.);
- Describe the operating conditions (mobility, communication etc.);

In the second activity (*Detail*), the SoS requirements are fully detailed. The results may include, for example, behavioral models of the SoS, e.g., statecharts (Harel, 1987). The foreseen actions include:

- Decide on the architecture according to the SoS classification (see Section 2.2.3) ;
- Define roles;
- Establish collaboration rules;
- Detail interaction mechanisms (events, messages, services, interfaces, protocols etc.);
- Model interactions among players (black boxes yet);

In the third activity (*Organize*), SoS requirements are grouped in such a way that each group of requirements describes a particular role and can be delivered to appropriate teams of systems engineers (suppliers). Different suppliers deploying systems assigned to a single role results in a FoS. This activity enables the separation of concerns and each supplier can focus exclusively on the requirements that its system must meet to compose harmoniously the SoS. Moreover, the information made available should be enough so that systems engineers are able to decide on the benefits of belonging to the SoS and how the purpose of their systems can be improved.

In Figure 23, the suppliers play the role of deploying candidate systems to compose a virtual SoS. Since neither a coordinator nor glue code is expected, implementation activities

like coding are performed exclusively by systems engineers. Each supplier manages its own system in an independent way, thus it decides if and when its system will belong to the SoS. To support such decisions, SoS engineers must provide enough information about “why” and “how”.

The SoS can be composed only when every role has at least one candidate system able to play it. More than one system can be invited to play the same role, for example, to increase redundancy and implement fault tolerance policies. In both cases, we are considering the existence of a mission to be achieved by players. However, a mission may not even exist in a virtual SoS. Indeed, SoS engineers can provide an environment in which collaboration rules and interaction mechanisms are comprehensive enough to enable emergent behaviors in large scale. In such environment, each member is free to continuously improve its purposes and extend its boundaries without being engaged in achieving supra-purposes, i.e., SoS goals. In this case, virtually any system that meets the SoS requirements is allowed to join it.

This thesis addresses both of the above scenarios in an evolutionary manner. A virtual SoS is firstly composed without a mission. Then, a mission is defined and roles are assigned to candidate systems so that the mission can be achieved by means of collaborations. This is done by carrying out the SoSE process of Figure 23 twice through the “*Evolution*” path.

3.4. An example

Figure 24 shows an example of how the proposed SoSE process of Figure 23 would apply to the military domain. This example is not intended to be developed in its entirety. The objective is linking the SoSE process to a real problem so that an SoS-based solution and its possible benefits can be discussed.

An Unmanned Aerial Vehicle (UAV) would overfly a bounded area to seek and destroy a target (mission). The SoS composition must be dynamic so that additional members can be added if necessary. For night and day missions, multiple cameras (one for each period) can be used to increase image quality. The SoS is decentralized, i.e., only critical and advisory information is controlled by a coordinator. The flight must be bounded by a specific area and must conform to regulatory laws. SoS members must communicate through a local network. Information about weather, location, and climate must be available to every member. When the mission is completed the UAV must go home.

The given description summarizes the information provided by SoS engineers during the *Sketch* activity. In the *Detail* activity, individual and collective behaviors are modeled and

assigned to particular players. The proper composition of the players' abilities, e.g., fly, seek, and shoot should result in the achievement of the SoS mission. Then, in the *Organize* activity, the detailed SoS requirements are grouped per role, e.g., carrier, seeker, and shooter, each associated to a distinct FoS, e.g., aircraft, camera, and ballistics respectively.

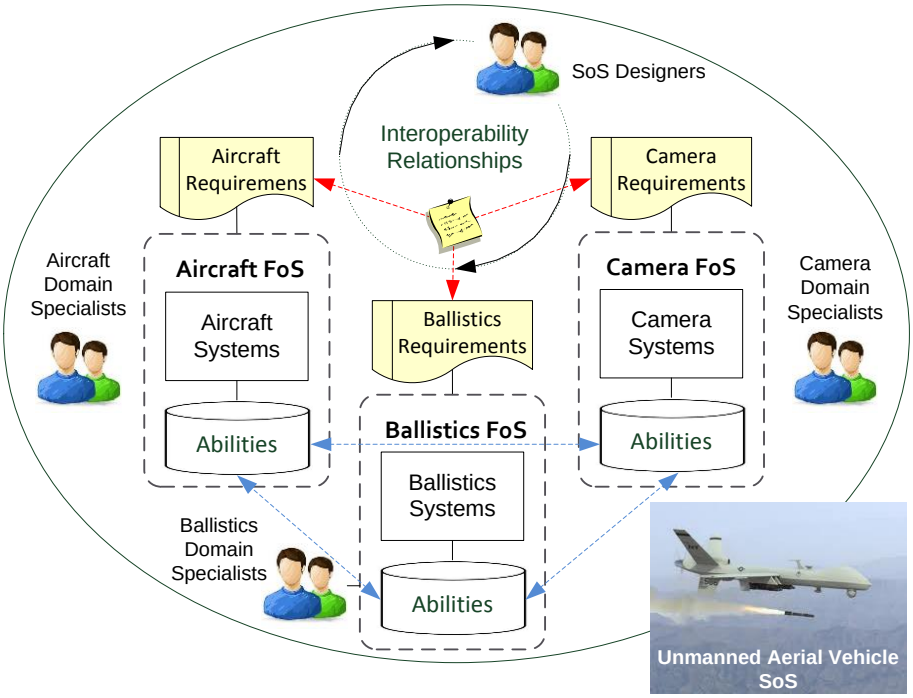


Figure 24. The proposed SoSE process applied to the military domain

If FoS repositories exist, then they are searched for suitable candidate systems. Those requirements with no matching systems are delivered to suppliers, which probably include teams of domain specialists. Each team can decide to evolve a legacy system, develop a new system, or deny the development. The resulting systems should be able to compose the SoS harmoniously.

The term "*ability*" is extensively used in this thesis and must be clearly understood. It is a high-level definition that distinguishes the system's capability of doing something useful from the way it is made available to other systems. In Figure 24, fly control, seek target and fire weapons are abilities of the aircraft, camera and ballistic systems respectively, which can be made available into the SoS through different interfaces and interaction mechanisms, e.g., agents or services.

Adopting an SoS-based solution for the problem above can enable several missions to be accomplished by simply choosing different candidate systems, e.g., aircrafts with different flight autonomy and payload capacity, cameras with different light sensitivity, and ballistics with different purposes and accuracy. The SoS requirements are similar to all those variants

and a minimal effort should be necessary to deploy them all. Moreover, completely different missions can be assigned to the SoS like the one described below.

Roberts and Walker (2010) described a competition in which a person supposedly lost in a desert region should be located and to whom a survival kit should be delivered. It would be declared winner the team able to accomplish this task by launching the kit from an UAV to the shortest distance from the person. Figure 25 illustrates the described scenario.

Figure 25. Competition scenario (Roberts e Walker, 2010)

The above missions can be both deployed from similar SoS requirements. However, they may require systems with particular abilities to achieve their goals, e.g., camera to either seek a military target or locate a person. This is why FoS is important in the SoS context. Indeed, the greater is the number of FoS members, the greater is the possibility of matching the required abilities for different scenarios. It is important to notice that such abilities are intrinsic to each system and deployed by system engineers. Thus, a camera will independently pursue its own purpose and still collaborate with other systems to achieve the SoS goal. It means that the camera can make decisions, e.g., asking other systems to abort the mission if it is not able to guarantee minimal accuracy after asking the aircraft to fly in different speeds and altitudes. The same applies to the ballistic system.

Considering conventional system-based solutions, the above problems would probably not be addressed in the same way. Changing the aircraft for alternative missions, for example, would require new software for the camera and ballistics compatible with the new hardware. This is particularly true, if all components are assembled and deployed as a single system. In this case, the resulting system is rarely evolutionary and will require great effort to be adapted to different missions as those presented above.

3.5. Related work

Our approach has some similarities with the service-oriented SoSE approach proposed by Lewis et al. (2011) (Figure 9). The requirements, matching, composition, testing and deployment phases are equally specified as well as the suppliers. The process is more specific than that we proposed and presents two major differences. The SoS developer role is supposed to make adjustments in the SoS so that services can be integrated and used. As discussed previously, we intend to deploy virtual SoS which is not supposed to have glue codes. Additionally, the process complies with SOA and services are gathered to achieve the SoS goal. In this case, member systems may not get benefits when belonging to the SoS since they can simply provide services. Finally, gathering services to achieve a goal may create dependency relationships in the SoS as in SOA. This should be prevented according to the autonomy characteristic of an SoS member.

Similar ideas of SoS have been explored by researchers as the emerging paradigm named “Internet of Things” (IoT) (Atzori, Iera, and Morabito, 2010). The basic idea of IoT is the presence of a variety of objects, such as RFID, NFC, sensors, actuators, mobile phones, etc. which, through unique addressing schemes are able to interact with each other. However, those “things” mostly do not conform to the characteristics of SoS constituent systems, e.g., autonomy. This is clear in the examples of IoT presented in the literature as the smart parking system (Libelium, 2013), which provides extensive parking management solutions to help motorists to save time and fuel. The solution comprises smart parking sensors buried in parking spaces to detect the arrival and departure of vehicles. It will allow the deployment of systems to book parking spaces directly from the vehicles. It sounds clear that emergent behaviors are not expected in such solution and that “things” can still maintain whole-part relationships.

3.6. Final considerations

In this chapter we proposed an SoSE process to support our objective of deploying a virtual SoS. It is intended to be a guide rather than a detailed process with enforced roles and assignments. It is similar to the SoSE approach of Figure 9 but it is more generic than that. Moreover, it can deploy SoS that fully conform to the SoS distinguishing characteristics (see Table 1). This makes it different from the approach proposed by Lewis et al. (2011) (Figure 9) even for service-oriented SoS.

The proposed SoSE process enables SoS engineers to group systems in families (FoS) according to the abilities they can share. Later, these systems can be reused to compose SoS and help on the achievement of more complex goals. Among the benefits, we expect to obtain improvements in the interoperability, reusability, and extensibility of conventional systems.

Because we aim to perform the composition of an SoS comprised of conventional systems, the proposed SoSE process pursue different objectives relative to those ones pursued by known research institutions, e.g., SEI (2014). Those institutions usually refer to constituent systems as large-scale, often complex, and information-intensive legacy systems, mostly belonging to military, civilian government and commercial domains (SEI, 2014). Thus, SoS problems commonly discussed in the literature are frequently complex and large in scope, e.g., interoperability and management issues, and their solutions are beyond the scope of this thesis. The vice-versa is also valid, i.e., we think that the design of conventional systems as SoS, e.g., smart homes, robots and unmanned vehicles can reveal problems and solutions currently not discussed in the literature. For example, the construction of FoS repositories may not make sense in the military domain where systems are usually complex and unique. The same applies to systematic development strategies since complex systems can hardly be developed from scratch to meet SoS requirements.

As already stated, the proposed SoSE process of Figure 23 was intended to be generic so that it can be applied distinctly to build SoS with different characteristics. For example, to build traditional SoS, the RE phase of the SoSE process could comprise a set of practices to support interoperable acquisitions of legacy systems (Smith II and Phillips, 2006). In the context of this thesis, systems will mostly be developed or reengineered to compose SoS, thus different RE practices may apply. Another example refers to the matching of compatible systems, which can vary significantly. When complex legacy systems are going to compose the SoS, this activity may not even be necessary since those systems are often unique. However, it might be very important, for example, when service-oriented or agent-based systems are considered. In these cases, the matching techniques probably will be different.

Chapter 4

A requirement engineering approach to SoS

4.1. Initial considerations

According to Meyers et al. (2006), traditional RE techniques do not appear to fully apply to SoS; they are necessary, but not sufficient. Their arguments include: **a)** Emergent behaviors result from interactions among multiple systems. Thus, it is not clear neither the origin of the requirements for those behaviors nor the way they are collected on individual systems; **b)** an SoS can be assigned to different missions. Each mission may meet the interest of a particular group which owns temporally the SoS requirements. In this context, the SoS requirements may be available only when a particular mission requires certain functionality from a particular set of member systems; and **c)** member systems can change individually at their own rates. Thus, the SoS capabilities are expected to evolve over time. This way, they can be only defined at any given instant. Those arguments suggest that new approaches to SoS requirements may be necessary.

Because of paradoxes among different types of SoS (see Section 2.2.2), we believe that a unique RE approach that applies generically to SoS could hardly be achieved. For example, paradoxes among directed and virtual SoS (Maier, 1998) affect their requirements differently. Indeed, directed SoS have a well-known purpose, capabilities and possibly constituent systems (see example of Figure 24), thus the SoS requirements can be described and agreed in a managed way. On the other hand, virtual SoS lacks a centrally agreed upon purpose and emergent behaviors may occur in large scale. Thus, all the above concerns apply.

Conventional RE approaches usually aim to describe a system as something concrete to be built from related and mostly interdependent parts that must fit well together to form an integrated whole, i.e., the system. In this thesis, we consider a virtual SoS an abstract system rather than a concrete system which may require different approaches to be described and composed. This can be better understood by using analogies as the one given in the following.

Consider a set of furniture (systems). Each furniture is a concrete object (the whole) built from interdependent parts (components) carefully designed to fit well together. Each part

as well as the whole has its own set of well-defined requirements. The whole usually defines the main purpose of the composition, e.g., a chair to sit, which hardly changes. Now, consider an empty room. It is an abstract concept to define a finite space bounded by concrete objects. Actually, the space and its properties are the only characteristics that can be assigned directly to the room, e.g., large and empty. Others may apply to walls, floor etc. as they are concrete objects that surround the room space, e.g., colors. Finally, the room (SoS) can be setup for different purposes, e.g., brainstorm room, classroom, and waiting room, by simply composing chairs (constituent systems) inside it in different numbers and arrangements. Each chair preserves its independence (independent systems) and they can be added to, removed from, or moved into the room freely. Also, they can evolve separately and uncoordinatedly, e.g., maintenance done by manufacturers whenever necessary. Different compositions are intended to achieve different results (SoS goal) which can evolve over time. For example, several new purposes can be assigned to the room as the quantity, variety and composability of objects inside it increase. Frequently, the requirements to compose the room can be only known when it has been assigned to a certain purpose.

According to the above analogy, designing furniture for a purpose is different from composing a room with furniture disposed in different ways for a variety of purposes. In this context, we can say SoS engineers are designers rather than developers. Indeed, they may even not know how to build concrete systems and still be able to design useful SoS with them. The scene-based RE approach we propose in this chapter is intended to support SoS engineers in describing SoS and their particularities; some of them were discussed above.

Before presenting the proposal, there is one more issue to be discussed involving SoS requirements; the SoS users. Drawing use cases is a well-known approach to discover users when designing conventional systems. Typically, users are described as actors that interact with the system and play particular roles (Booch, Rumbaugh, and Jacobson, 1999). Then, those users and roles hardly change. In the SoS scenario this might be different. Using the above analogy, a user could be a person who sits in a chair. In this case, he/she is a user of a constituent system, i.e., the chair, and plays the role of sitting (be there). However, when the same user sits in a chair of a classroom, the role he/she can play changes automatically. This is because he/she can now interact. This possibility enables the user to get extra benefits, e.g., learning. Moreover, if he/she sits in a brainstorm room, his/her role changes significantly. Thus, although users of constituent systems and their roles are well-known, in the SoS context

they can only be fully defined for a given instant. Moreover, some roles may only exist in the SoS context, e.g., teaching.

It is important to notice that getting extra benefits might be the reason why users of constituent systems are motivated to participate of and contribute to the SoS (the *Belonging* characteristic of Table 1). Thus, this must be addressed carefully in SoSE.

This chapter is organized as follows. In Section 4.2, we discuss related work. In Section 4.3, we propose a scene-based RE approach that enables engineers to describe an SoS by means of roles, collaboration rules and interaction mechanisms. In Section 4.4, we present a case study that describes the sample virtual SoS to be deployed. Finally, in Section 4.5, we present our final considerations.

4.2. Related work

As discussed previously, an SoS can evolve over time, so accurate assumptions about it can only be done at any given instant (Meyers et al., 2006). Thus, eliciting and describing relevant instants of the SoS operation is crucial. A possible solution to accomplish this task is dividing the SoS goal in small but yet meaningful instants and then explore the set of possible scenarios for each instant. Huynh and Osmundson (2006), for example, use this approach to model SoS using SysML diagrams (Figure 26). Their SoS integrates systems to counter terrorism emanating from the maritime domain. The problem description statement leads to the definition of use cases to be incorporated into use case diagrams, which support the development of scenarios (Sutcliffe, 2003). A use case specifies a sequence of actions a system performs. A use case diagram shows the relationships among actors, the system, and use cases.

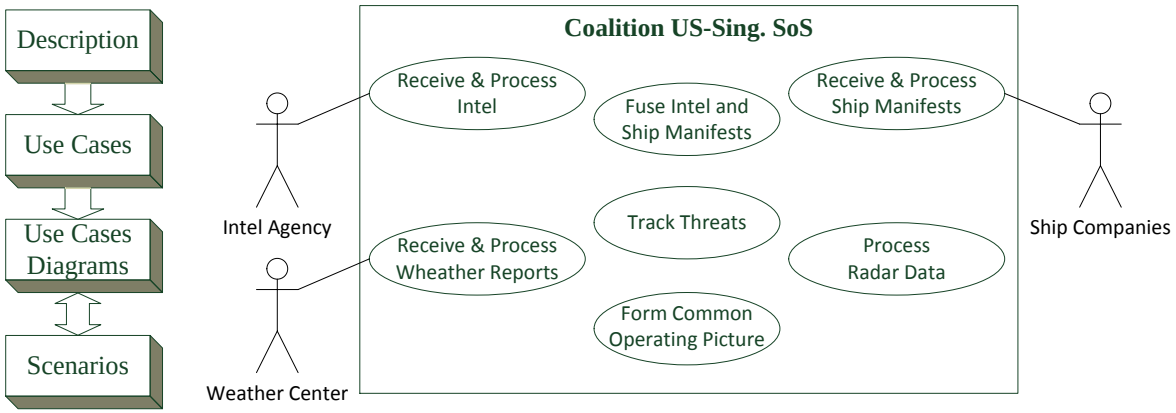


Figure 26. Coalition US-Sing. SoS (Adapted from Huynh and Osmundson, 2006)

Later, SoS constituent systems are arranged in a class diagram as illustrated in Figure 27. The <<system>> and <<external> stereotypes indicate systems belonging to the SoS and external systems respectively. Compositions group similar systems distributed geographically, for example, Singapore Radars is composed of Radar 1, Radar 2, and Radar 3.

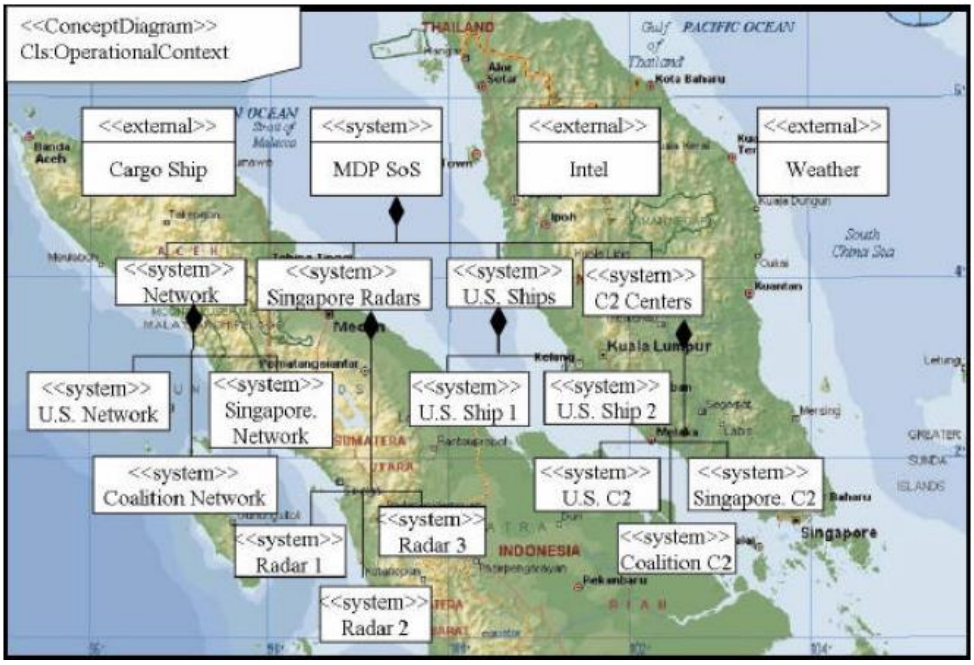


Figure 27. SoS concept and context diagrams (Huynh and Osmundson, 2006)

Finally, a role is assigned to each system (a block) and interactions (flow of control and data messages among blocks) are modeled by means of activity and sequence diagrams.

The described approach for SoS appears to conform to conventional component-based development (Heineman and Councill, 2001) and reinforces the assumption from those that consider traditional SE processes necessary and sufficient to support SoSE. It is important to notice that this RE approach fits well our SoSE process described in Section 3.2 and shown in Figure 23.

We discussed previously the paradoxes among different types of SoS and said that a unique SoS RE approach could hardly be applied generically. The given example emphasizes that. Remembering Mayers et al. (2006), because of the SoS dynamism, its capabilities can only be defined in terms of what it can do at any given instant. Thus, eliciting and describing relevant instants to the SoS should be the basis of SoS RE approaches. We think that use cases like those of Figure 26 may not be appropriate to this purpose. Indeed, an instant may comprise several systems interacting to produce a unique collective behavior which is part of the SoS goal. Different scenarios can emerge from each instant, for example, when important

capabilities are missing or emergent behaviors takes place. The use case diagram of Figure 26 captures actions like “Fuse Intel and Ships Manifest”, which say little about a relevant instant or a collective behavior.

The lack of RE approaches to elicit and describe all the relevant instants of the SoS operation and the different scenarios they comprise motivated us to propose a RE approach, which is presented next.

4.3. The proposed scene-based RE approach

To elicit and describe the SoS operation in terms of relevant instants and to explore the scenarios they comprise, we propose a scene-based RE approach. It derives from an analogy with a movie, which has a similar problem and an accepted solution in the real world.

A scene is the smallest meaningful unit of a movie. A movie is made of a sequence of scenes arranged properly. Each scene is responsible for revealing part of the whole movie story to the audience. In a scene, there may be actors acting both individually and collectively. A scene ends up only when its purpose has been fully accomplished.

By analogy, the SoS goal (movie) can be described in terms of scenes (meaningful behaviors) which are conducted neatly by the SoS members (actors). Each scene encloses an observable behavior that must be fulfilled collectively by the SoS members to achieve the SoS goal. It is important to distinguish scenes from scenarios (Sutcliffe, 2003) which, in this case, can be used to explore different situations in a particular scene and to give them appropriate solutions. This way, the possibility of getting expected behaviors in the scenes increases and the SoS has a better chance to succeed.

4.3.1. Overview of the approach

Figure 28 illustrates the proposed scene-based RE approach to SoS. It complies with the discussed topics and fits well the proposed SoSE process of Figure 23. It was intentionally kept simple and does not impose specific SE techniques for eliciting the requirements of the SoS and its members.

The proposed RE approach was divided in three phases (vertical lanes of Figure 28). In the first phase (High-Level Requirements - SoS) SoS engineers provide a clear definition of the SoS goal, which abilities are necessary to achieve it and what are the possible candidate systems. General requirements like security and operational rules must be provided as well.

In the second phase (Middle-Level Requirements - Scenes) SoS engineers divide the SoS goal in scenes (relevant instants) and define, for each scene, if, when and how each

member system will compose it and how they should interact. Variations in the composition and interactions lead to different scenarios that must be identified and fully described. Then, all requirements are refined up to the level they can support system engineers to evolve their systems to SoS members. Before the SoS requirements are deployed they are organized so that only pertinent information is delivered to each team of system engineers.

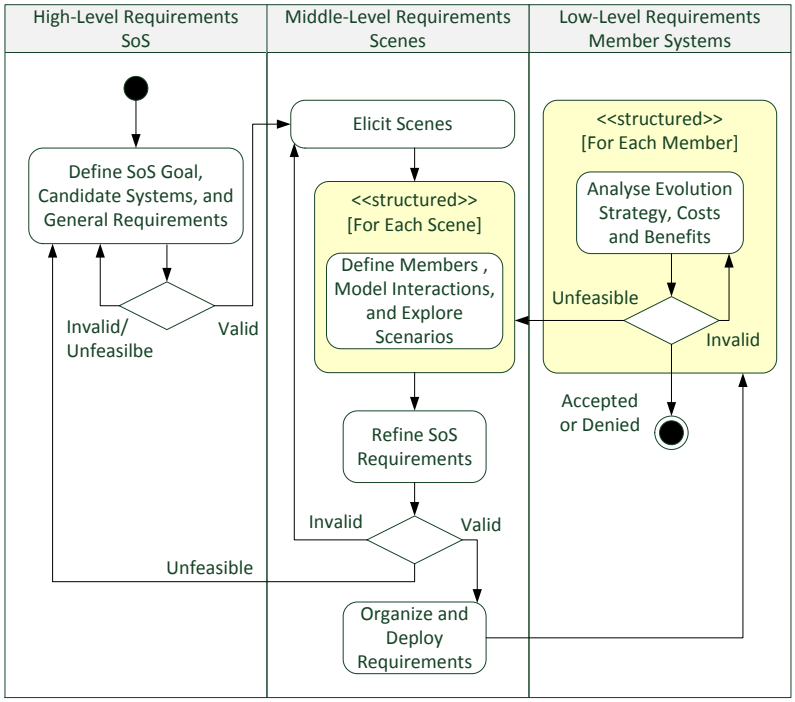


Figure 28. Scene-based RE approach

Finally, in the third phase (Low-Level Requirements - Systems), system engineers analyze the best strategy to evolve their systems to SoS members according to the given requirements. Moreover, they analyze the abilities that their systems should provide and the benefits they will have when belonging to the SoS. These analyses support a decision making about feasibility. If unfeasible, the requirements are reviewed by the SoS engineers until the indicated problems are solved. Otherwise, the resulting artifacts are used to develop the SoS members. Notice that systems engineers can also decide to not evolve their systems based, for example, on cost-benefit analysis.

4.3.2. Example

Considering the SoS of Figure 24 the following scenes could be defined: 1. Take Off; 2. Reach delimited area; 3. Seek target; 4. Destroy target; 5. Go home; and 6. Land. These scenes are clear about the SoS goal and its relevant instants. At this point they do not refer to any particular system but give insights about the abilities necessary to accomplish them.

Notice that scenes enable engineers to describe the SoS behavior in a manageable way. Different from use cases, they give a clear idea of what should happen to the SoS to succeed. Moreover, scenes can easily indicate alternatives to fault conditions. For example, if Scene 2 fails, i.e., the delimited area cannot be reached (possible scenario), Scene 5 can be forced to abort the mission. Observe that since Scene 5 is an expected SoS behavior, the involved systems must know exactly how to behave to carry it out. For example, the ballistic and camera systems can enter in standby mode and the aircraft can change the flight coordinates to go home. Those behaviors must all be modeled in the second phase of the proposed RE approach, i.e., the Middle-Level Requirements – Scenes phase. However, SoS engineers must design interactions in such a way that the SoS members can recognize what scene they are currently in, and when and how to move to another one. This could be done, for example, by modeling scenes as SoS states and define mechanisms to trigger transitions coordinately. Figure 29 exemplifies this approach. It shows that if any intermediate scene fails, then the constituent systems collaborate to go home. However, each scene can be assigned to different sets of systems, and make them all to collaborate and move through the scenes coordinately requires well-designed interactions.

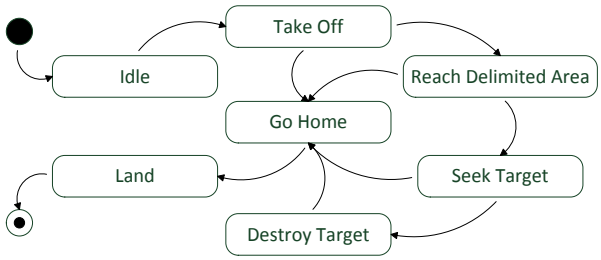


Figure 29. SoS scenes modeled as states

Another important issue to discuss about scenes is their individuality. Each scene may have a completely different complexity level, thus different compositions of the constituent systems can be defined to accomplish them. This way, SoS engineers can decide, for example, to assign a coordinator to complex scenes and let the systems operate freely in others. This affects directly the current concept of SoS type (see Section 2.2.3). In the given example, the SoS could alternate between open and close types or even virtual.

Finally, it must be clear that no assumption is done about SE techniques to compose the proposed SoS RE approach. For example, SoS engineers can decide to model the SoS of Figure 24 using feature models (Kang et al., 1990) as illustrated in Figure 30.

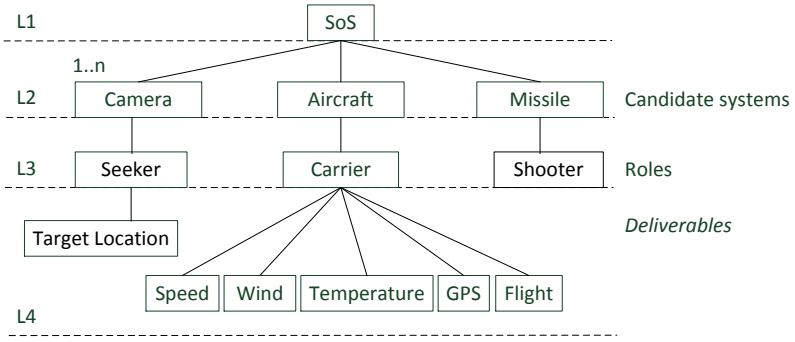


Figure 30. SoS feature model

In the given example, the model was divided hierarchically in four layers {L1,...,L4}. The SoS was placed at the top layer (L1). L2 contains the candidate systems, i.e., aircraft, camera and missile. L3 contains the roles that the candidate systems are intended to play, i.e., carrier, seeker, and shooter respectively. Finally, L4 contains the deliverable capabilities that can be shared among systems. This approach can be used, for example, to support the first phase of the proposed RE approach, i.e., the High-Level Requirements - SoS phase (Figure 28).

4.3.3. The scene-based approach and the proposed SoSE process

In this section we discuss how the proposed scene-based RE approach of Figure 28 fits the proposed SoSE process of Figure 23. The activities performed in the High-Level requirements - SoS phase of the RE approach match entirely those required in the *Sketch* phase of the SoSE process, i.e., outline the SoS goal, assign the mission to different players with specific roles, decide on the rules etc. (see Section 3.2). A possible set of resulting artifacts and their relationships are shown in Figure 31.

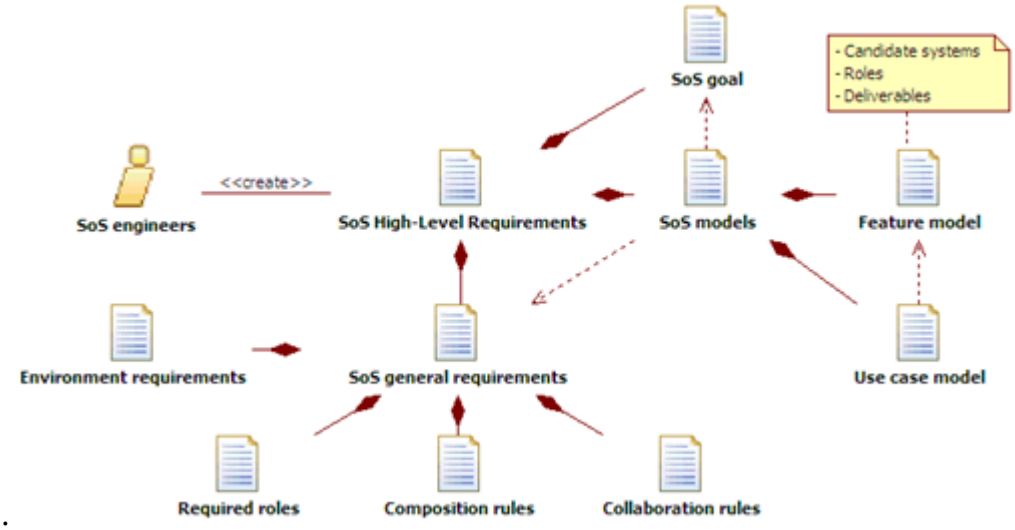


Figure 31. Artifacts of the High-Level Requirements - SoS phase

The activities of the *Detail* and *Organize* phases of the SoSE process are performed during the Middle-Level requirements - Scene phase of the proposed RE approach. A possible set of resulting artifacts and their relationships are shown in Figure 32.

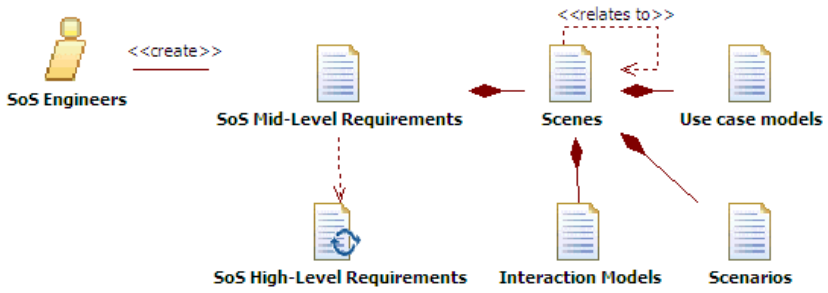


Figure 32. Artifacts of the Middle-Level requirements phase

Finally, the activities of the Low-Level requirements - Member Systems phase of the RE approach are intended to be performed by the suppliers in the SoSE process.

4.4. Case study – Describing the sample virtual SoS

The sample SoS defined is this section aims to state our argument that even small and simple conventional network-enabled systems can share abilities and contribute to SoS goals. It will be evolved gradually as the next chapters are presented to show how the proposed SE approaches fit the generic SoSE process of Figure 23 in a complementary way. The resulting SoS must be evolutionary and conform to the SoS distinguishing characteristics described in Table 1, i.e., autonomy, belonging, connectivity, diversity and emergence.

The first sketch of what the SoS should do, what should be the constituent systems, what they should do, and how they should do it, is described in the Addendum. Having an agreed version of this sketch was the first big challenge and showed the difficulty of shifting to new paradigms. Most of the time we spent on describing it, was attempting to break chains. Questions like: “What is SoS about?”, “Why it is not about SOA or Components?”, and “Why using SoS to solve traditional problems?” had to be answered, mostly with examples using analogies as the one presented in Section 4.1.

Next, the proposed scene-based RE approach of Figure 28 is applied as part of the generic SoSE approach of Figure 23 (*Sketch* and *Detail* phases) to describe the sample virtual SoS and its properties.

4.4.1. High-Level requirements – SoS

In this section the artifacts proposed in Figure 31 are used to describe the High-Level requirements of the sample SoS. It is important to notice that the set of resulting artifacts may change depending on the RE approaches used.

SoS goal

Systems will compose the SoS to share calculation services, thus they can collectively increase their computational capabilities and provide to their particular users a larger number of calculations than they are currently able to perform when operating standalone. Whenever available, systems will also publish updated information gathered locally to and only to those systems that declared explicit interest in receiving them.

SoS general requirements (GR)

○ SOS-GR#1: Systems must be independent and be able to run consistently in and out of the SoS boundaries.

○ SOS-GR#2: Systems can decide by themselves to participate or not of the SoS. Thus, no assumptions must be made about their availability.

○ SOS-GR#3: Systems can evolve freely and in an uncoordinated way.

Environment requirements (ER)

○ SOS-ER#1: The network infrastructure necessary to support the communication among SoS members must be available and operate reliably so that these systems can share abilities while belonging to the SoS.

○ SOS-ER#2: Temperature must not cause the systems to freeze or overheat. Ideal conditions comprise values between 5°C and 40°C.

○ SOS-ER#3: Moisture must not cause circuits and components to rust or leakage currents between nets. Ideal conditions comprise values between 35% and 65%.

Required roles (RR)

- Calculator: System able to do calculations;
- Publisher : System able to gather and publish information;
- Processor : System able to receive information and deliver services.

○ SOS-RR#01: SoS members can play more than one role concurrently.

○ SOS-RR#02: SoS members can start/stop playing a role in an uncoordinated way. But, they must notify all other members about their decisions.

Composition rules (CR)

○ SOS-CR#1: The number of member systems is variable.

○ SOS-CR#2: The composition occurs dynamically.

○ SOS-CR#3: Systems can enter/leave the SoS freely and in an uncoordinated way.

○ SOS-CR#4: Whenever a system enters/leaves the SoS, it must announce itself and inform all the abilities, i.e., calculation and information, it is making available/unavailable to all SoS members.

○ SOS-CR#5: Whenever a system enters the SoS, it must query all SoS members for their abilities currently available.

○ SOS-CR#6: Whenever a system needs to stop sharing an ability it informed previously as being available, it must announce itself and inform that the ability is becoming unavailable to all SoS members.

○ SOS-CR#7: Whenever a system needs to restart sharing an ability it informed previously as being unavailable, it must announce itself and inform that the ability is becoming available to all SoS members.

○ SOS-CR#8: The received data in SOS-CR#4, SOS-CR#5, SOS-CR#6, and SOS-CR#7 must be saved locally to support future interactions with the sender system.

Collaboration rules (CR)

○ SOS-CL#1: The SoS operation must be maintained without a coordinator (virtual).

○ SOS-CL#2: The rules for sharing calculation and information must be clearly defined.

○ SOS-CL#3: Native calculation must be executed preferably related to shared calculation.

○ SOS-CL#4: Native calculation must become available to all SoS members (shared).

○ SOS-CL#5: Publications must be forwarded only to subscribers.

○ SOS-CL#6: Publications must be forwarded on a regular basis.

Feature model

The feature model of Figure 33 models hierarchically the sample SoS being described, the candidate systems, and their roles and deliverables. Particularly in this case, the roles are not specific like those of Figure 30 and can be played by a great number of systems. Thus, no candidate system is specifically defined at this point and they were represented by a '?' in the feature model. This can be changed later when, for example, a mission is assigned to the SoS and it requires specific publishers like GPS systems and processors like cameras. This

highlights the evolutionary characteristic of SoS and how the proposed RE approach can handle it.

The information represented visually in the feature model of Figure 33 enables the SoS general requirements previously described to be extended as follows:

○ SOS-GR#3: Shared calculations comprise addition (+), subtraction (-), multiplication (*), division (/), addition in memory (M+), subtraction in memory (M-) and memory read (MR).

○ SOS-GR#4: Shared calculations are optional and a system may not share any (according to the multiplicity OR {0..5} assigned to the role calculator).

○ SOS-GR#5: At least two systems must compose the SoS (according to the multiplicity {2..*n*} assigned to the SoS).

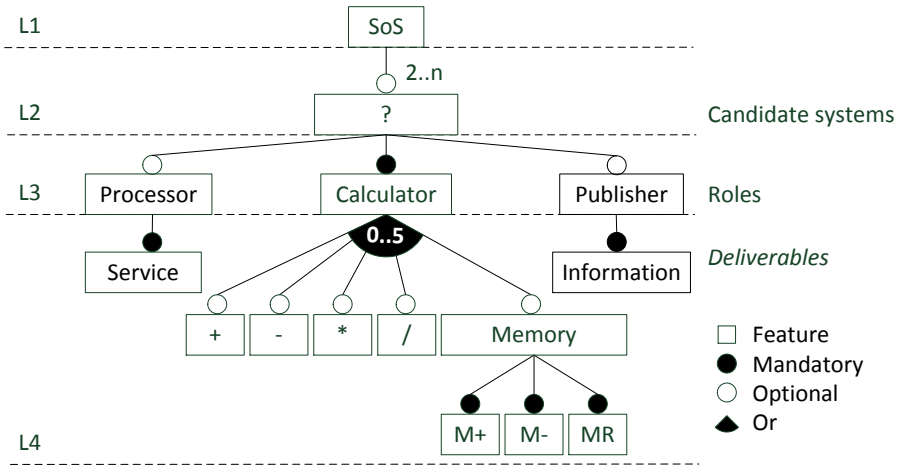


Figure 33. Sample SoS feature model

Use case model

In the context of this thesis, SoS is rather an abstract system composed of concrete systems (the room with chairs discussed in Section 4.1). SoS members and their users, not the SoS, actually perform actions that result in observable behaviors, which are then assigned to the SoS as a whole. However, at this point, SoS members are still black boxes and the only information we know about them are the roles they can play. Moreover, SOS-RR#1 states that a system can optionally play more than one role concurrently (see the '?' candidate system of Figure 33). Thus, defining uses cases as usual may be confusing for the sample SoS and we can fall in the situation where there are no concrete systems to assign actions and no optional action can be precisely assigned to a system. This scenario is exemplified in Figure 34. The *Publish Shared Information* use case cannot be precisely assigned to a system because it is

optional. Also, we do not know what concrete system is going to compose the SoS to play this role. Moreover, because of the SoS dynamism, the players of this role can change over time.

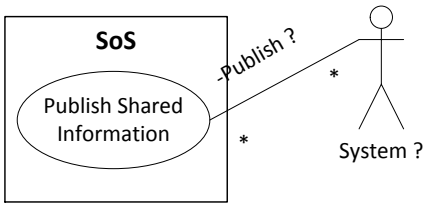


Figure 34. A use case of the sample SoS

To solve this problem we built a use case diagram for the sample SoS in an unusual form. Figure 35 shows the resulting diagram.

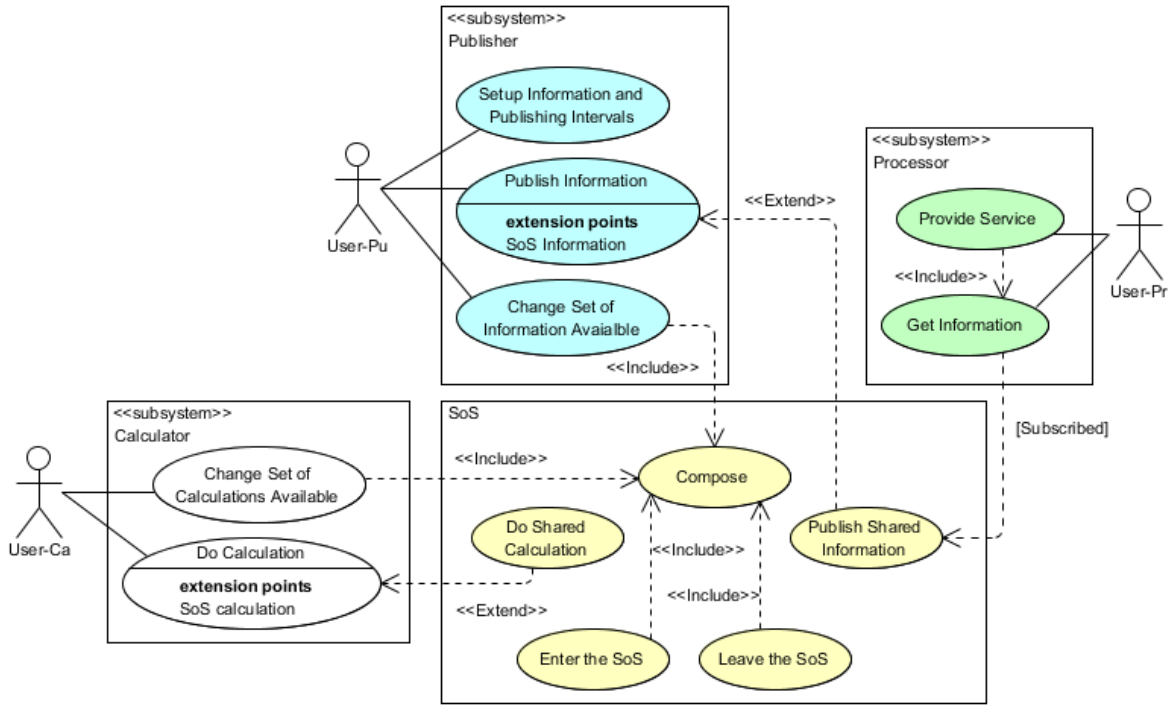


Figure 35. The sample SoS use case diagram

Roles were modeled as subsystems. Actions were not assigned to any member system since they are still unknown. Actors are people involved with the SoS, e.g., users of member systems. The rule that applies is: if a member system exists, whatever it is, and it can play a certain role, thus consider the embodied use cases of this role (a subsystem) as part of the system's use cases. This enables the reuse of the roles (and their use cases) throughout the SoS and avoids the need for duplicating them in every system that plays a certain role. This model can describe systems playing a variable number of roles as well. Moreover, it enables the description of the SoS use cases without defining any specific SoS member.

Notice that the use cases belonging to a subsystem apply to the systems that enclose that subsystem even when these systems are running out of the SoS, i.e., autonomously. In the SoS, these use cases describe actions that contribute to the SoS goal no matter the systems that carry them out. Finally, the use cases belonging to the SoS describe the actions performed collectively by the SoS members via interactions. Indeed, the SoS as an abstract system does not perform any action. These uses cases are summarized in the following:

UC - Compose

Description:

In the SoS composed of a set of Systems $S = \{S\text{-}n \mid n \in \mathbb{N}^* \wedge n \text{ is finite}\}$, the composition of the systems occurs dynamically. This collective behavior may vary due to three types of events that happen in an uncoordinated way: a) a system enters the SoS; b) a system leaves the SoS; and c) the deliverables made available by a member system change.

Participants: All member systems

Pre-conditions: Systems apt to do compositions, i.e., meet SOS-CR# requirements

Post-conditions: SoS composition concluded successfully

Successful scenario:

1. A member system $S\text{-}i \in S$ broadcasts it is going to either enter or leave the SoS, or its set of deliverables initially announced has changed;
2. Every member $S\text{-}j \in S–\{S\text{-}i\}$ reacts properly to the event according to the composition rules defined to the SoS (see requirements SOS-CR#).

Steps 1-2 repeat whenever one of those three events occurs

UC – Do (Shared) Calculation

Description:

In the SoS composed of a set of Systems $S = \{S\text{-}n \mid n \in \mathbb{N}^* \wedge n \text{ is finite}\}$ and a subset of Calculators $C \subseteq S$ that enable a set of operations $O = \{+,–,*,/,M+,M–,MR\}$, the User-Ca of a Calculator $C\text{-}i \in C$ can perform a set of native calculations $O_N \subseteq O$ as well as a set of shared calculations $O_S \subseteq O–O_N$ shared by other Calculators through interactions. In this scenario, a new set of calculations $O' \mid O' \cap O = \emptyset$ can emerge natively from the set of calculations currently available in the SoS.

Participants: Native - Calculator $C\text{-}i \in C$

Shared - Distinct pair of Calculators $C\text{-}i \in C$ and $C\text{-}j \in C–\{C\text{-}i\}$

Pre-conditions: At least one calculation available, i.e., $O_N \cup O_S \neq \emptyset$

Post-conditions: Calculation performed successfully

Successful scenario:

1. User-Ca chooses a (new) calculation and sets its parameters;
2. The calculation is performed successfully by either C-*i* (native) or C-*j* (shared)
3. The result is presented to User-Ca;

Steps 1-3 repeat until User-Ca has performed all the desired calculations.

Extensions:

2a. The calculation fails

1. User-Ca is notified
2. The last calculation state is recovered
3. Starts Step 1

The above use case describes a seeming chaotic scenario in which Calculators request concurrently and continuously to other Calculators to do calculations not available natively. Considering systems dynamically entering and leaving the SoS in an uncoordinated way, relationships are continually being created and destroyed and a Calculator may often have to request the same calculation to different Calculators (redundancy). New calculations can emerge natively anywhere at any time, e.g., ℃ ↔ ℉, %, etc. This is how the SoS (whole) can be larger in scope than the sum of the abilities of its members (parts) individually.

UC – Publish Shared Information

Description:

In the SoS composed of a set of Systems $S = \{S\text{-}n \mid n \in \mathbb{N}^* \wedge n \text{ is finite}\}$ and a subset of Publishers $P \subseteq S$, every Publisher P-$i \in P$ will periodically gather and publish information to all those members interested in receiving it, i.e., a set of Subscribers $S_B \subseteq S - \{P\text{-}i\} \mid S_B \neq \emptyset$ that explicitly submitted a subscription request directly to the Publisher P-*i*.

Participants: Publisher P-*i* and a set of Subscribers S_B

Pre-conditions: Publisher able to publish information (see SOS-CL# requirements)

Post-conditions: Information published to every Subscriber in S_B

Successful scenario:

1. User-Pu sets the information to be published and the publishing interval;
2. Publisher gathers updated information;
3. Publisher forwards the information to the Subscribers;
4. Subscribers receive the information and process it appropriately;
5. Publisher waits the publishing interval.

   Steps 2-5 repeat until the Publisher leaves the SoS

Extensions:

2a.There is no updated information available or it was not set to be published

1. Starts Step 5

3a. Publishing failed

1. Starts Step 5

Up to this point all the SoS distinguishing characteristics described by Boardman and Sauser (2006) (see Table 1) are being complied. **Autonomy**: members are fully independent

and can operate normally out of the SoS; **Belonging**: all members get benefits of belonging to the SoS; **Connectivity**: flexible composition and collaboration rules grant prolific interactions among systems; **Diversity**: a great number of systems can compose the SoS to increase diversity; and **Emergence**: a countless number of emergent behaviors can arise from shared abilities.

Disruptive situations can be prevented, e.g., by improving redundancy. Whenever a system leaves the SoS, the abilities it is currently sharing is probably being shared by other systems as well. The lack of a central management authority and a coordinator to maintain the SoS operational, the absence of an agreed mission to be accomplished and emergent behaviors characterizes the sample SoS as virtual (see Section 2.2.3).

Next, the sample SoS goal described in this section is divided in scenes, which are fully described separately to demonstrate how the SoS requirements can evolve incrementally.

4.4.2. Middle-Level requirements – Scenes

Actions performed collectively, e.g., those described by the use cases of Figure 35 belonging to the SoS, can comprise either only one type of interaction, e.g., "Do Shared Calculation", or multiple types, e.g., "Compose". In the second case, a system may be required to behave differently depending on the action it performs or the perceived actions performed by other systems. For example, a system operates distinctly when it enters or leaves the SoS, and also, when it perceives other systems entering or leaving the SoS it belongs to. This way, modeling collective actions like “Compose” in a single step may be complex and result in confusing descriptions and diagrams. To ease this activity, we assigned a different scene to represent each type of interaction. Figure 36 illustrates the scenes obtained when this approach was applied to the sample SoS. They are detailed next.

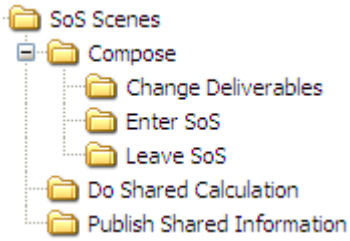


Figure 36. Scenes of the sample SoS

Notice that in the sample SoS, scenes will not happen sequentially like in the example of Figure 29. Instead, Compose, Do Shared Calculation and Publish Shared Information will happen concurrently during the SoS operation.

Scene 1	**Compose/Enter SoS**

The SoS composition changes dynamically because a new system is entering the SoS.

Requirements

○ SOS-SC1#1: A system must announce itself to all SoS members (*broadcast*) whenever it enters the SoS.

○ SOS-SC1#2**:** When entering the SoS a system must:

1. Inform all other members what abilities it can share, e.g., calculation, publishing;
2. Query all other members about the abilities they can share;
3. Enable the use of shared abilities locally.

○ SOS-SC1#3**:** Whenever a system is notified about a new system entering the SoS that can share abilities still not available locally, it must enable the use of those shared abilities. An internal association must be maintained linking those shared abilities and the new system.

○ SOS-SC1#4**:** Whenever a system is queried about the abilities it can share, it must send back that information immediately, directly, and exclusively to the requester.

○ SOS-SC1#5**:** Whenever a system receives a subscription request, it must send back the confirmation and the most updated information immediately, directly, and exclusively to the requester. Otherwise, the requester may keep unattended for a long time if the publishing interval has been set too high.

In the use cases that follow, the symbol ➔ represents an interaction point, which often indicates information exchange among systems, e.g., events and messages.

Use case model

Figure 37. Scene 1 – Compose/Enter SoS

UC – Enter the SoS	UC – Compose
New System	***SoS Members***
Pre-conditions:	
• System ready to enter the SoS	• Systems belonging to the SoS
Successful scenario:	
1. Announces itself to SoS members and informs what abilities it can share ➔	2. Detects a system entering the SoS and starts sharing those of its abilities that are not available locally yet. 3. If the system publishes useful information, then sends a subscription request back to it ➔
4. Receives a subscription request and updates the list of subscribers.	
5. Sends the confirmation and the most	6. Handles published information.

updated information back to the requester ➔	
7. Queries SoS members about the abilities they can share ➔	8. Receives a query about shared abilities. 9. Informs the native abilities being shared➔
10. Makes available those shared abilities that are not available locally yet.	
11. If the SoS member publishes useful information, then sends a subscription request back to it ➔	12. Receives a subscription request and updates the list of subscribers 13. Sends the confirmation and the most updated information back to the requester ➔
14. Handles published information.	
Extensions:	
1a. Does not share any ability 1. Starts Step 7	2a. No new system detected 1. Use case ends 2b. All the shared abilities are already available locally 1. Starts Step 8 3a. No information published or not useful 1. Starts Step 8
4a. Subscription request times out 1. Starts Step 7	
7a. Does not required shared abilities 1. Use case ends	8a. Query times out 1. Use case ends 9a. Does not share any ability 1. Use case ends
10a. Query times out 1. Use case ends 10b. All the shared abilities are already available locally 1. Use case ends	
11a. No information published or not useful 1. Use case ends	12a. Subscription request times out 1. Use case ends

Scene 2 **Compose/ Leave SoS**

The SoS composition changes dynamically because a member system is leaving the SoS.

Requirements

○ SOS-SC2#1**:** A system must announce itself to all SoS members (*broadcast*) whenever it leaves the SoS.

○ SOS-SC2#2**:** When leaving the SoS a system must:

1. Inform all other members what abilities are currently being shared (and will stop)
2. Disable shared abilities locally

○ SOS-SC2#3**:** Whenever a system is notified about other system leaving the SoS that hosts abilities shared locally, it must disable the use of those shared abilities. The internal association linking shared abilities and the other system must be cleared.

Use case model

Figure 38. Scene 2- Compose/Leave SoS

UC –Leave the SoS	**UC – Compose**
System that leaves	***SoS Members***
Pre-conditions:	
• System about to leave the SoS	• Systems belonging to the SoS
Successful scenario:	
1. Announces itself to SoS members and informs it is leaving ➔	2 Detects a system leaving the SoS and makes unavailable the abilities shared by that system 3. If the system is subscribed for receiving information, unsubscribes it.
4. Disables shared abilities locally	
Extensions:	
	2a. No system leaving the SoS 1. Use case ends

Scene 3 **Compose/Change Deliverables**

The SoS composition changes dynamically because a member system changed its set of shared abilities.

Requirements

○ SOS-SC3#1**:** Whenever shared features are being enabled, this scene must behave similar to Scene 1 / Steps 1..14 to inform all SoS members about them.

○ SOS-SC3#2**:** Whenever shared features are being disabled, this scene must behave similar to Scene 2 / Steps 1, 2 to inform all SoS members about them.

Scene 4 **Do Shared Calculation**

A shared calculation is performed by a system on behalf of other system

Requirements

○ SOS-SC4#1**:** Every system able to perform calculations must share this ability with all SoS members.

○ SOS-SC4#2**:** Calculations can be either stateless, e.g., * and /, or statefull, e.g., MR. In the second case, partial results are stored in the host system and can be retrieved any time.

Use case model

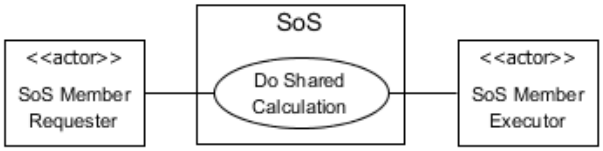


Figure 39. Scene 4 - Do Shared Calculation

UC – Do Shared Calculation	**UC – Do Shared Calculation**
Requester System	***Executor System***
Pre-conditions:	
• System belonging to the SoS • Calculation available	• System belonging to the SoS • Calculation shared
Successful scenario:	
1. Gets parameters and requests calculation ➔	2. Receives calculation request.
	3. Performs calculation.
	4. Returns result to the requester ➔
5. Receives calculation result and treats it	
Extensions:	
1a. Native calculation 1. Does calculation locally 2. Starts Step 5	2a. No pending calculation request 1. Use case ends 2b. Calculation could not be done 2. Treats error 3. Starts Step 4 2c. Calculation is statefull 1. Recovers last state 2. Does calculation locally 3. Saves current state 4. Starts Step 4

Scene 5 **Publish Shared Information**

Information is published to subscribers

Requirements

○ SOS-SC5#1**:** The publishing interval is an intrinsic property of each Publisher.

○ SOS-SC5#2**:** Information must be published only when it differs from the last published information.

○ SOS-SC5#3**:** Subscribers can unsubscribe anytime for certain information. Whenever necessary, a new subscription request can be submitted.

○ SOS-SC5#4**:** Whenever a Publisher receives duplicated subscription requests, i.e., belonging to an existing subscriber, it must ignore that request. However, it must send back immediately, directly and exclusively to the requester the most updated information.

Use case model

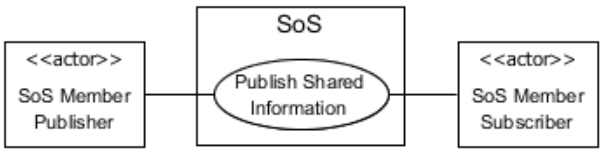


Figure 40. Scene 5 – Publish Shared Information

UC – Publish Shared Information	**UC – Publish Shared Information**
Publisher System	***Subscriber System***
Pre-conditions: • System belonging to the SoS • Information available • At least one subscriber in the list	 • System belonging to the SoS • Already subscribed
Successful scenario:	
1. Gathers information when the publishing interval times out 2. Publishes information to subscribers ➔	
	3. Receives and treats published information. 4. Requests to unsubscribe ➔
5. Receives a request to unsubscribe and removes the requester from the list of subscribers.	
	6. Requests to subscribe (first time) ➔
7. Receives a request to subscribe and adds the requester to the list of subscribers. 8. Sends updated information ➔	
	9. Receives and treats published information 10. Requests to subscribe (duplicated) ➔
11. Receives a duplicated request to subscribe and ignores it. 12. Sends updated information ➔	
	13. Receives and treats published information.
Extensions:	
1a. Publishing interval did not time out 1. Use case ends 2a. Information has not changed 1. Use case ends 5a. Requester is not a subscriber 1. Ignores request to unsubscribe 2. Starts Step 7	
	9a.13a. Information times out 1. Use case ends

When the description of the scenes and the interactions they comprise ended, we started evolving the SoS requirements according to the proposed scene-based RE approach of Figure 28. The level of detail should be enough for the requirements to be understood clearly by systems engineers. This way, their systems can be developed or maintained more naturally to compose the SoS. A challenge in this phase was to define suitable mechanisms to support interactions. Our decision was to design a service-oriented SoS because of the platform-independent nature of services. However, it was necessary to figure out how services could meet the existing SoS requirements like dynamic compositions and collaborative work. For these purposes WS-* standards have been adopted (see Section 2.4.1).

The refined requirements of the sample SoS are presented in the following.

Communication

○ SOS-RF#1**:** The WS-* standards must be used to support service-oriented interactions among systems.

○ SOS-RF#2**:** WS-Discovery messages, e.g., *Hello*, *Probe*, and *Bye* must be sent *broadcast* via UDP, port 2048. Every SoS member must verify for messages at this port periodically.

○ SOS-RF#3**:** Publishing must be done *unicast* via TCP, port 1025. Every Subscriber must verify for information at this port periodically.

○ SOS-RF#4**:** Calculations must be done *unicast* via TCP, port 1024. Every Calculator must verify for requests at this port periodically.

Composition

○ SOS-RF#5**:** Every ability, e.g., *, / and MR has a unique ID which is bitwise thus they can be concatenated in the WS-Discovery messages to reduce traffic.

○ SOS-RF#6**:** A *Hello* message must be used by every system to announce itself to all SoS members when it enters the SoS (*broadcast*). Additionally, this message must inform the set of abilities the system can share (array of bitwise IDs) and the system's address (*EndPoint*).

○ SOS-RF#7**:** The *Hello* message must be sent even if the system entering the SoS has nothing to share at that moment.

○ SOS-RF#8**:** *Hello* messages must be used by SoS members to announce new shared abilities becoming available either for the first time or after maintenances.

○ SOS-RF#9**:** The *EndPoint* address is unique for each SoS member and must be used for direct interactions.

○ SOS-RF#10**:** A *Bye* message must be used by every system to announce itself to all SoS members when it leaves the SoS (*broadcast*). Additionally, this message must inform the set

of abilities the system is currently sharing (array of bitwise IDs) and the system's address (*EndPoint*).

○ SOS-RF#11**:** The *Bye* message must be sent even if the system leaving the SoS is not sharing abilities at that moment.

○ SOS-RF#12**:** *Bye* messages must be used by SoS members to announce shared abilities becoming unavailable, for example, for maintenances.

○ SOS-RF#13**:** The same ability can be shared concurrently by different systems. They can be queued by member systems to increase redundancy and fault tolerance.

○ SOS-RF#14**:** The *Probe* message must be used to query all member systems (*broadcast*) about the abilities they can share. It can optionally inform the specific set of abilities the system has interest (array of bitwise IDs) and must inform the system's address (*EndPoint*).

○ SOS-RF#15**:** *ProbeMatch* messages must be sent (*unicast*) in response to queries (*Probe*) whenever a system can share abilities that match those informed in the *Probe* message. If the abilities do not match the message must not be sent. The response time must not exceed 5s.

○ SOS-RF#16**:** Every SoS member must maintain a particular list of shared abilities and their respective *Endpoints* based on the *Hello*, *Bye*, and *ProbeMatch* messages.

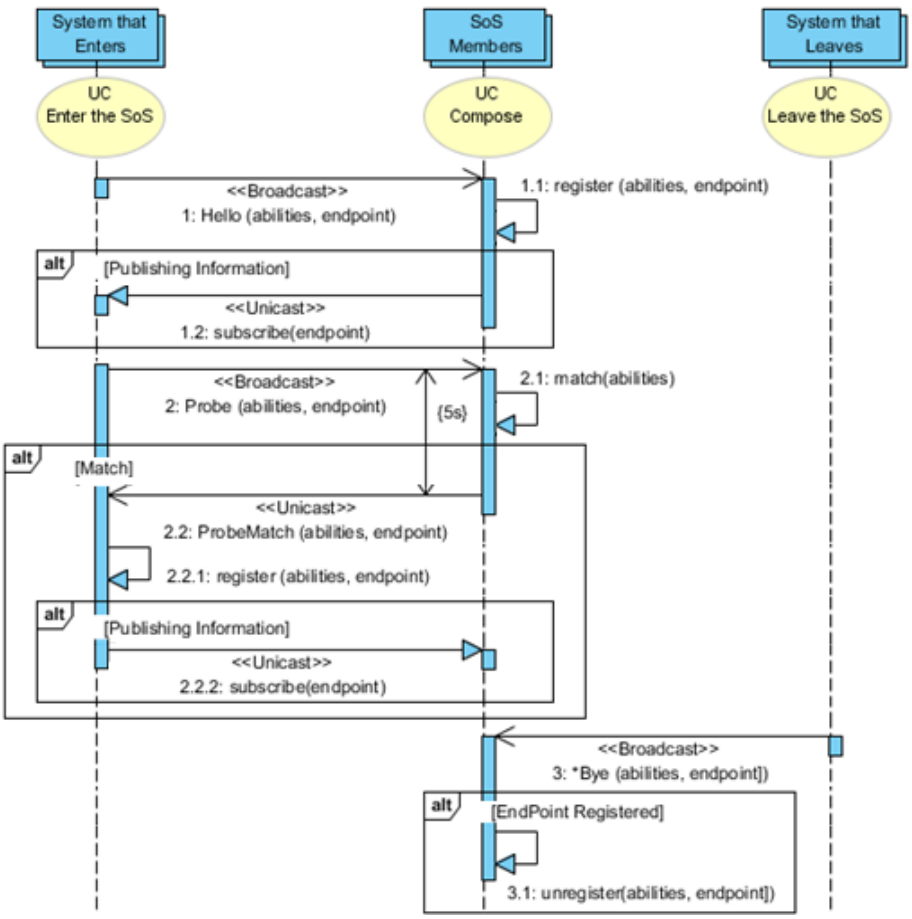


Figure 41. SoS composition sequence diagram

The sequence diagram of Figure 41 represents visually the above uses cases related to the dynamic SoS composition.

Abilities

○ SOS-RF#17**:** Every shared ability must be described in the SoS service contract.

○ SOS-RF#18**:** A member system may eventually not share any ability. However, it can still benefit from the abilities shared by other members.

4.5. Final considerations

The proposed scene-based RE approach aims to help SoS engineers to evolve the SoS goal and generic requirements to more detailed requirements of the member systems in terms of interactions. The flow of activities (see Figure 28) was aimed to be as generic as possible thus it can be instantiated to comprise different RE techniques like scenarios and features.

The approach introduced the concept of scene as a small but yet meaningful collective behavior that contributes to the SoS goal, which can be described as an arrangement of scenes (Figure 36). A scene can comprise one or more member systems and represent a simple or complex task. Finally, scenes enable the SoS development to be performed incrementally, i.e., one scene at a time. We believe that this can reduce the complexity of SoS projects and contribute to increase the rate of success.

The case study we performed (Section 4.4) revealed both strengths and weakness of the methodology we used to elicit SoS requirements based on traditional SE approaches, e.g., use cases and UML diagrams. For example, the way we described interactions by placing related use cases side by side and indicating with ➔ whenever an interaction occurs is similar to the approach proposed by Tian et al. (2011). They called the vertical lanes of entity containers, which comprise distinct but related logic. Entities can interact through interfaces, which are connected by interaction logics placed in the border space of the containers. This space was called interaction space (Figure 59). This approach is discussed in Section 5.4. It is particularly useful to separate concerns. Indeed, when the use cases are finished, each lane isolates a particular logic (of a member system or FoS) and its interaction interface. This is exactly what is necessary to conclude the second phase (Middle-Level Requirements - Scenes) of the proposed scene-based RE approach (Figure 28), i.e., organize and deploy requirements. When this is done, the requirements phase of the proposed SoSE approach (*Sketch*, *Detail* and *Organize*, Figure 23) ends.

The weakness of this approach relates to complex interactions involving several systems. Although this is not the case of the case study we performed, it is clear that it would be difficult to place all systems' logic (use cases) side by side to elicit all interaction points. Moreover, the interaction space would become confusing. A possible solution to this problem is proposed in the next chapter.

Chapter 5

A modeling approach to SoS

5.1. Initial considerations

Before discussing modeling approaches to SoS it is important to distinguish between studying interoperability mechanisms among systems and modeling interactions. The first intends to identify and classify different forms of interoperability among systems, and to propose solutions to overcome the barriers to achieve it (Morris et al., 2004). Studying interoperability mechanisms is out of the scope of this thesis. On the other hand, modeling interactions among systems intends to build interaction models to help engineers, for example, to design collective behaviors. A typical model resulting from this activity is shown in Figure 42 (Jennings, 2001).

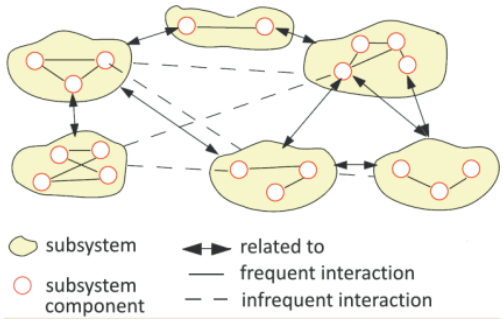


Figure 42. View of a canonical complex system (Jennings, 2001)

Modeling systems' compositions by means of visual elements can probably increase their understanding and support discussions about interactions before they are effectively designed, possibly reducing risks and costs associated to systems' misbehaviors discovered in advanced stages of the development process. Statecharts (Section 2.5) can describe dynamic scenarios where states of a particular system evolve concurrently and coordinately by reacting to stimuli from the system's environment. In the SoS context, compositions comprise interactions among systems that run independently. In this case, each system runs in its own environment and its states can evolve by reacting to stimuli from other system's environment as well.

We are motivated by the possibility of using statecharts to model SoS. Specifically, we want to use statecharts to support the modeling of systems' interactions in the second phase of

the scene-based RE approach we proposed in Section 4.3 (see Middle-Level requirements – Scenes, Figure 28). The problem of using uses cases for this purpose was already discussed in Section 4.5.

Although several variants have been proposed to statecharts since they were presented, most of them aim to overcome problems in the original formalism (von der Beeck, 1994). Thus, statecharts still lack completeness to represent systems' compositions. Specially, they lack notions of multiplicity (of systems), and interactions and parallelism (among systems). In this chapter, we propose extensions to statecharts to solve these problems.

The extensions we propose to statecharts are symbolic notations rather than textual that aim to visually improve the modeling of interactions among systems and increase their understanding. They are intended to be used since the very beginning of the development processes involving multiple integrated systems to build abstract models of these systems and to represent visually their interactions. These models can help engineers, for example, to identify interaction points (Tian et al, 2011), predict communication bottlenecks (Kotov, 1997), design collective behaviors, and compose SoS (Brownsword et al, 2006).

This chapter is organized as follows. In Section 5.2, we propose statechart extensions to support notions of multiplicity of systems, and interactions and parallelism among systems. Also, we discuss the improvements achieved when those extensions are applied to systems, events, and states. In Section 5.3, we exemplify the use of the proposed extensions to model concurrent Skype™-like systems able to perform conference calls. In Section 5.4 we discuss relate work. In Section 5.5 we evolve the case study started in Section 4.4 by modeling the sample virtual SoS using statechart extensions. Finally, in Section 5.6, we present our final considerations.

5.2. The proposed statechart extensions

In this thesis, we use the terms transition and interaction distinctly. Transition refers to a direct relationship between two states of a system. It is triggered by an event from the system's environment and may cause the active state to transition to a new state or eventually to the same state. On the other hand, interaction refers to an indirect relationship among concurrent states of a system or related states of different systems. It is triggered by an action and causes the broadcast of an event to either inside or outside the boundaries of the source system. In the target systems this event may trigger transitions.

In Figure 43, we exemplify transitions and interactions and their use in the SoS context. The suffixes *a* (action), *i* (interaction), and *e* (event), like in ω*a*, ω*i* and ω*e*, are used to prevent ambiguities. Systems 1 and 2 run independently and some of their states relate via the interactions ω*i* and λ*i*. The initial state configuration of the composition is (E,A;C) (';' separates groups of states from different systems).

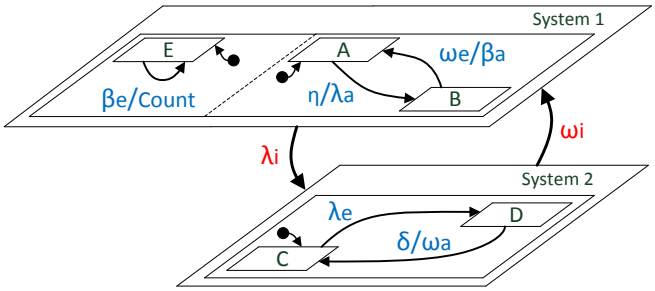


Figure 43. Transitions and interactions in the SoS context

In System 1, an event η triggers the transition A→B and generates an action λ*a*. λ*a* triggers an interaction λ*i* that causes the broadcast of an event λ*e* to other systems. In System 2, λ*e* triggers the transition C→D and the current state configuration changes to (E,B;D).

In System 2, an event δ triggers the transition D→C and generates an action ω*a*. ω*a* triggers an interaction ω*i* that causes the broadcast of an event ω*e* to other systems. In System 1, ω*e* triggers the transition B→A and generates an action β*a*. β*a* triggers an implicit interaction β*i* (omitted) and causes the broadcast of an event β*e* to the orthogonal states, i.e., state E (as in Figure 21). Finally, β*e* triggers the transition E→E, the cycle is counted, and the current state changes again to (E,A;C).

To keep the notations clear during the analysis phase, the whole interaction process, i.e., action, interaction, and event, can be represented simply by an event broadcast (Harel, 1987) among systems. For example, λ*a*, λ*i*, and λ*e* of Figure 43 could be represented simply by λ in both System 1 and System 2. λ*i* could be omitted. However, they should be used later in the development phase as a reference to the details of each part of the interaction process separately. For example, λ*i* and ω*i* could refer to different interaction mechanisms, such as services (Lewis et al., 2011) and agents (Jennings, 2001).

In the SoS context, environmental elements, e.g., events and data, are not shared among systems naturally. In the given example, System 2 is not aware of the counting being performed by System 1. This is because the event β and the counter data are intrinsic elements of System 1. Certainly, these elements can be shared through proper interactions.

Statecharts lack completeness to describe systems' compositions like that of Figure 43. Indeed, the notions of multiplicity and parallelism relate primarily to states, e.g., overlapping and orthogonality respectively (see Section 2.5). This constraint is not new. Actually, it was discussed by Harel (1987) since the very beginning of statecharts. However, he left the solution incomplete by sketching a statechart with ideas to be evolved. This statechart is shown in Figure 44.

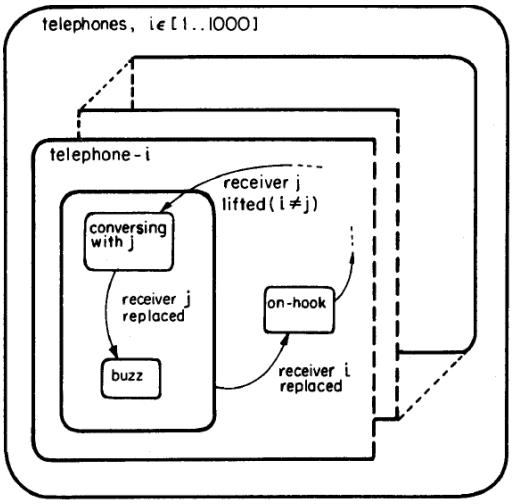


Figure 44. Phone call statechart (Harel, 1987)

Harel's statechart of Figure 44 sketches a phone call, i.e., a composition between a caller and a receiver system. The result is a 3D diagram that tends to comprise a limited number of systems and interactions. In fact, its complexity can increase fast, for example, if the statechart is extended to model a conference call involving several telephones. The statechart fully exemplifies the scenario we are interested in modeling more clearly. It highlights the constraints of the current visual elements to support the modeling of systems' compositions. Overlapping states are not fully represented as well as systems' interactions, e.g., the "…". An example is the hidden overlapping state "conversing with i" in telephone-j. Moreover, "receiver j replaced" is not an event perceived directly by telephone-i. Instead, it results from an interaction between the systems, particularly in this case, after telephone-j goes "on-hook" (the "…").

The lack of notion of parallelism among systems makes it difficult to represent compositions in a plain diagram. Additionally, the lack of notion of multiplicity of systems and actions hinders, for example, the representation of how many systems are affected by an event/action and how many systems share a particular overlapping state. Also, the lack of notion of interaction makes difficult the modeling of relationships among states of different systems (the "…").

Despite these constraints, Figure 44 allowed us to have insights into a solution to model systems' compositions: 1) compositions can be described more clearly if we consider one "Observer" at a time, e.g., Observer-*i* for telephone-*i*; 2) interactions not involving the observed system can be hidden without compromising its understanding and must only be presented to appropriate observers; 3) there must be elements to represent the interactions coming from/going to other systems different from the observed system.

Next, we propose a set of symbolic notations to systems' interactions towards a solution that comprises all the items described above. They result from an analogy with multi-layer Printed Circuit Boards (PCB) (Khandpur, 2005), which are widely used by the electronic industry to connect electronic components through multiple circuit layers. Figure 45 illustrates internally a PCB with 3 layers. A *Through hole* goes all the way through the board and connects all layers. *Blind via* and *Buried via* connects two or more, but not all, layers of the board.

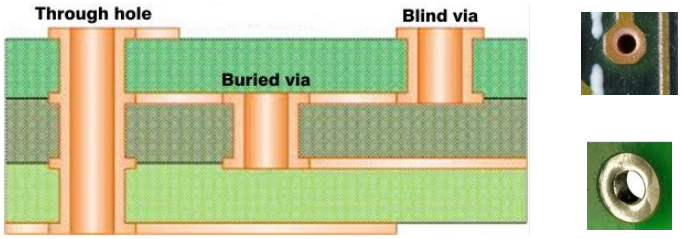


Figure 45. Multi-layer printed circuit board

In our analogy, each layer (a system) contains a set of components (states) connected by circuits (transitions). Sets of components from different layers are connected by holes (interactions). Signals (events) flow through circuits and holes concurrently and coordinately (relationships) thus the components can accomplish a common goal collectively. This scenario just describes a systems' composition. From the point of view of an observer, holes extend the observed layer with the notions of multiplicity (of layers), and interactions and parallelism (among layers).

Next, we present the proposed symbolic notations for holes (interactions) that represent what the observer sees when looking to a layer of the PCB. They are:

① The *Uni* notation (filled hole + '1') means that the hole connects the current layer with exactly one different layer of the board. In this case, it is filled because what the observer sees is the surface of some internal layer.

ⓜ The *Multi* notation (filled hole + 'm') means that the hole connects the current layer with at least one, but not all, different layers of the PCB.

◯ The *Broad* notation (empty hole) means that the hole connects all the layers, i.e., goes from one side to the other of the PCB.

Finally, the absence of any of these notations means no hole, i.e., no connection with other layers of the PCB.

5.2.1. Statechart extensions

The statechart extensions we propose to model systems' interactions result when the symbolic notations presented in the previous section, i.e., *Uni*, *Multi* and *Broad* are attached to statechart elements, specifically to states, events, and systems (comprehensive blobs). Next, we present these extensions and discuss how they improve statecharts with the notions of multiplicity (of systems), and interactions and parallelism (among systems). Because of the analogy with PCB, we will use the term PCB-statecharts to refer to statecharts that comprise one or more of the proposed extensions.

5.2.2. Statechart extensions applied to systems

Some statecharts extensions result when the proposed symbolic notations are attached to systems. They can be particularly useful to represent visually the overlapping of multiple systems that behave similarly or to highlight the uniqueness of a particular system. The resulting notations extend statecharts with the notions of multiplicity and parallelism of systems. Table 4 relates textual and symbolic notations for sets of systems. It considers a finite set of systems S = {A-n | $n \in \mathbb{N}^* \wedge n$ is finite}, a system A-$i \in$ S, and a set of overlapped systems O (see Table 4) related to A-i. The notion of multiplicity and parallelism is given by the tuple (S, A-i, O), where A-i represents the visible layer to an observer.

Table 4. Statechart extensions attached to systems

	Notations	
	Textual	Symbolic
Broad	$O = S - \{A\text{-}i\}$	*A-i* ◯
Multi	$O = \{ A\text{-}j \in S - \{A\text{-}i\} \mid P(A\text{-}j) \}$	*A-i* ⓜ
Uni	$O = \{ \}$	*A-i* ①

P(A-j) is an assertion that is true whenever A-i overlaps A-j ($i \neq j$). It enables an observer i of A-i to know which is the subset O of overlapping systems. This is particularly important to describe compositions that changes dynamically or that involves a particular subset of systems. If P(A-j) is always true, then O = S−{A-i} and A-i overlaps all systems (*Broad*). If P(A-j) is always false, then O = { } and A-i is unique (*Uni*). Otherwise, P(A-j)

defines a particular subset *O* of overlapping systems. Notice that the elements of the subset *O*, i.e., the systems A-*j*, can vary over time if P(A-*j*) describes a dynamic scenario, for example, similar systems A-*j* entering/leaving an SoS in an uncoordinated way. In this case, P(A-*j*) can only describe a subset *O* of overlapping systems at a given moment.

In this thesis we highlight the potential of using assertions P(A-*j*) to detail *Multi* ⓜ symbolic notations in PCB-statecharts and to improve notions of dynamism. However, it is out of the scope of this thesis to discuss assertions in depth since there must be numerous different scenarios and techniques to describe them. We left this topic open for future research.

Assigning an assertion to a *Multi* notation is optional. In Table 4, for example, if no assertion is assigned to A-*i*ⓜ, this notation will represent a generic subset of overlapping systems. Otherwise, it will represent precisely the subset of overlapping systems described by P(A-*j*). For more precise descriptions, languages like the OMG Object Constraint Language (OCL, 2012) can be used. For example, *Tuple* {*S*: *Sequence* {*A-1..A-4*}, *System*: *A-1*, *O*: *Set* {*A-2*, *A-4*}} can be used to describe a system A-*i*, which overlaps the systems A-*2* and A-*4* in a set of four systems A-*1* to A-*4* (Figure 46). Assertions can be set apart from the PCB-statechart or be part of it, for example, as UML notes like in Figure 46. In the following example, A-*3* is unique.

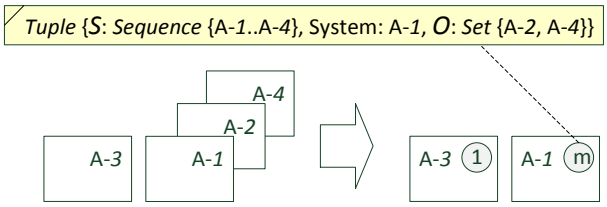


Figure 46. Using OCL to describe overlapping systems

A Private Automatic Branch eXchange (PABX) system is commonly used to make connections among the internal telephones of a private organization. Some of the following examples use a PABX to exemplify systems' interactions.

The PCB-statechart of Figure 47 shows a PABX system connecting two phones (*i* and $j \mid i \neq j$) to handle a call. The uniqueness of each system, i.e., Phone-*i* (caller), Phone-*j* (receiver), and PABX is represented by the *Uni* notation ① attached to them. In this high level statechart, the blobs are systems rather than states. Internal states were intentionally hidden. Just like λi and ωi of Figure 43, the arrows connecting the systems represent the interactions among them.

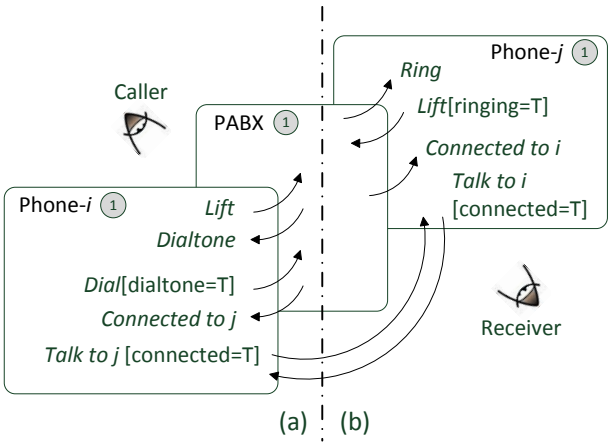


Figure 47. Example of symbolic notations attached to systems

According to the discussion started in this chapter, using observers to represent different perspectives of a systems' composition might be a good practice to improve clarity in the resulting model. In Figure 47, interactions are perceived differently by two distinct observers: the Caller (Figure 47a, left side) and the Receiver (Figure 47b, right side).

From the perspective of the caller, a successful scenario would mean: 1) the caller lifts the handset, 2) hears the dial tone from the PABX and 3) dials an extension number. The call is completed by the PABX and 4) the caller talks to the receiver (Figure 47a). Concurrently, from the perspective of the receiver, a successful scenario would mean: 1) the receiver hears a ring tone and 2) lifts the handset. The call is completed by the PABX and 3) the receiver talks to the caller (Figure 47b).

Despite of defining observers, the modeling of systems' compositions may still result in complex and confusing diagrams as the number of systems and interactions grows. In this case, identifying arrows would become a puzzle.

According to the analogy with PCB presented in Section 5.2, each observer should observe a particular system. Thus, in the previous example, an observer should be observing either a Phone system or a PABX system, but never both as in Figure 47. However, in this case, we could not use arrows to represent interactions because only one system is visible at a time. To solve this problem the proposed symbolic notations can be attached to events.

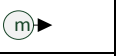
5.2.3. Statechart extensions applied to events

The major goal of creating symbolic notations to extend events in statecharts is improving the ability to represent systems' interactions without the inconveniences caused by the use of arrows, discussed previously. One may say that statecharts already provide such capability by means of event broadcast. Nevertheless, it is equally inefficient to visually

describe interactions in the context of this thesis. In fact, if broadcast notations are used and only one system is visible at a time, an observer may not be able to distinguish between local and broadcast events. The same happens if all involved systems cannot be represented together in the same model. For example, in Figure 21, looking to only one state, either the left or the right one, it is not possible to conclude that *b* is a broadcast event. This makes difficult, for example, to perform a visual impact analysis involving the event *b*.

Table 5 shows the statechart extensions that result when the proposed symbolic notations are attached to events. The resulting notations represent the holes (interactions) that an observer sees when looking to a layer (system) of a PCB (composition). Thus, even looking to a system at a time and without using arrows it is possible to observe interactions with a certain level of detail. For example, it is possible to identify interaction points and analyze if and how an event affects other systems. This ability would help engineers to create, model, and validate collective behaviors comprising several systems and interactions. In the SoS context, for example, they could exercise different candidate systems and interactions to achieve the SoS goal before making decisions.

Table 5. Statechart extensions attached to events

Notations		
	Outgoing	Incoming
Broad		
Multi		
Uni		

In Figure 21, incoming and outgoing signals (events) are placed respectively at the left and right side of the separator '/'. Thus, the notation ω/λ is clear in representing ω as an incoming event and λ as an outgoing event. This way, it would be enough to attach the proposed symbolic notations to events to improve the notions of multiplicity, interaction, and parallelism. However, an optional short arrow can be used to emphasize outgoing (arrow entering the hole) and incoming (arrow leaving the hole) events as shown in Table 5. Hereafter, we will always include them in the PCB-statecharts.

Using terms from the computer networking area, the interactions represented in Table 4 were named broadcast, multicast, and unicast. They visually represent interactions of a source system with all, many, or one (target) system respectively as exemplified in Figure 48. The broadcast of an event to orthogonal states can still be represented by simply not attaching any extension to it.

Figure 48. Broadcast, multicast and unicast interactions

It is important to notice that Multicast events can also be detailed by assertions P(A-*j*) in PCB-statecharts as discussed in Section 5.2.2 (see Table 4). In this case, P(A-*j*) can describe the subset *O* of target systems that will receive the event. It does not mean, however, that every target system A-*j* will handle the event. This will depend on the current state of each system.

Finally, P(A-*j*) can improve notions of dynamism as well. For example, it can describe the subset *O* of subscribers in publish/subscribe relationships, which can vary over time. However, it is not our intention in this thesis to discuss all possible multicast scenarios and techniques to describe them. This topic remains opened for future research.

Variations of the symbolic notations of Table 5 can be used, for example, to solve ambiguous situations. This approach is exemplified in Figure 49. An empty arrow is used to emphasize that the PABX (unique because of the notation ①) forwards a ring event to a system (Phone-*j* - the receiver) other than the one that originated the dial (Phone-*i* | $i \neq j$ - the caller).

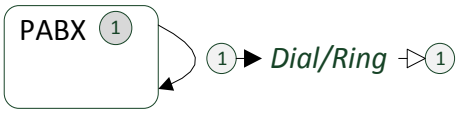


Figure 49. Solving ambiguous situations

The two PCB-statecharts of Figure 50 represent the scenario of Figure 47 with more details. Numeric identifiers were added only to help us on pointing to specific elements of interest. They do not belong originally to PCB-statechart diagrams. The symbolic notations of Table 5 were used instead of arrows to represent interactions among systems.

The notion of hierarchy also applies to the proposed symbolic notations. For example, Phone-*i* ○ means that all phone systems are implicitly represented in the diagram. They overlap because of their similarity. By hierarchy, the broad notation ○ applies by default to all internal states unless explicitly specified. In this case, it represents that they are all overlapping states and that every Phone-*i* implements them.

Each observer is now associated with a particular system because of the analogy with PCB. It is important to notice that to a Phone-i observer (Figure 50a), the Phone-i system can

be either a caller or a receiver. Therefore, Phone-i is modeled with both roles represented together. However, they are not concurrent, i.e., not orthogonal.

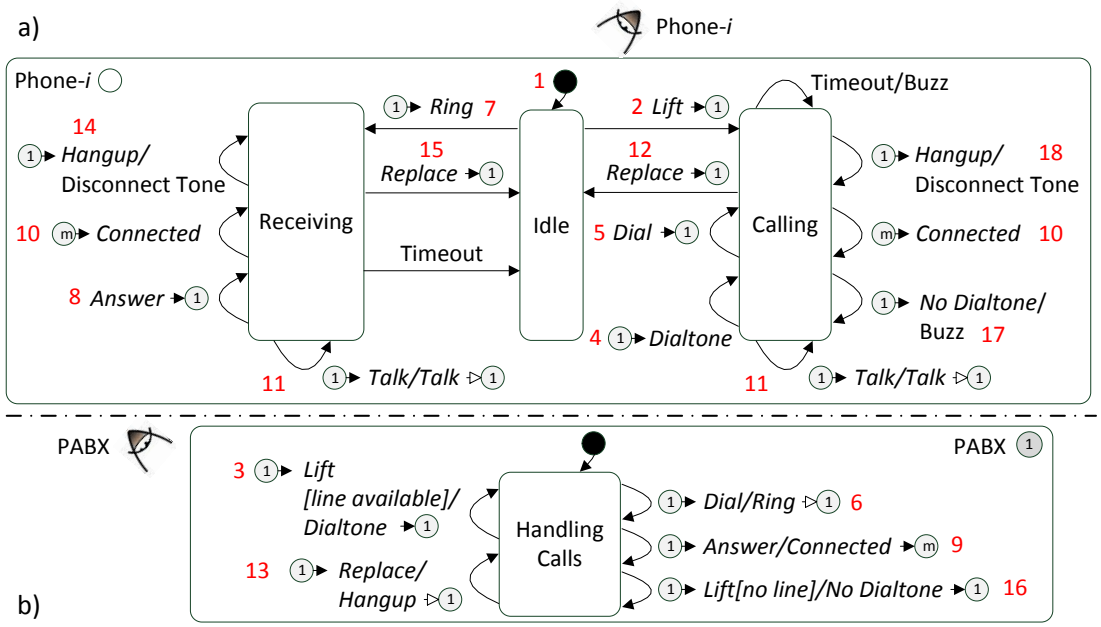


Figure 50. Example of symbolic notations attached to events

A successful scenario could be described by a caller as (use the numeric identifiers of Figure 50 as reference):

1. Phone-*i* system is *Idle* (default)
2. The handset is lifted to start a call.
4. A Dialtone is heard
5. An extension number is dialed
10. The call is completed
11. The conversation starts
12. The handset is replaced to end the call

Additionally, the same scenario could be described by a receiver as:

1. Phone-*i* system is *Idle* (default)
7. A Ring is heard
8. The handset is lifted to answer the call.
10. The call is completed
11. The conversation starts
14. The call hangs up
15. The handset is replaced

Finally, the same scenario could be described by PABX as:

3. A handset is lifted. Then, a Dialtone is generated.
6. An extension number is dialed. Then, a Ring is generated.
9. A call is answered. Then, a connection is established.
13. A handset is replaced. Then, the related connection is closed.

Notice that if the above scenarios are merged, the result is a collective behavior that supports a common goal, i.e., enabling two people to talk. This complies with our objective of modeling SoS using PCB-statecharts. Indeed, when the SoS goal is defined, candidate systems can be modeled separately, interactions can be designed to support collective behaviors, and the whole SoS operation can still be described by merging individual behaviors.

Some of the previous steps deserve further explanation.

Step 3: Two filled arrows mean that the incoming and outgoing interactions occur with the same system. In this case, the PABX system receives the event Lift caused by the lift of the handset in the caller system (Step 2) and, then, returns a dial tone signal to the same system resulting in Step 4.

Step 5: The dial activity is hidden since it is irrelevant to the analysis of the composition. Later, it could be detailed separately from the analysis model by unclustering (Harel, 1987) the *Calling* state as shown in Figure 51. This might be a good practice to increase the understanding of the analysis model by keeping the focus on the interactions.

In Figure 51, a dial starts when a key {0,…,9} is pressed (1). The triggered transition is directed to the proper state by the selection connector Ⓢ (Harel, 1987) according to the key pressed. This process repeats (2) until a delay of 10 seconds since the last key was pressed (3) is detected.

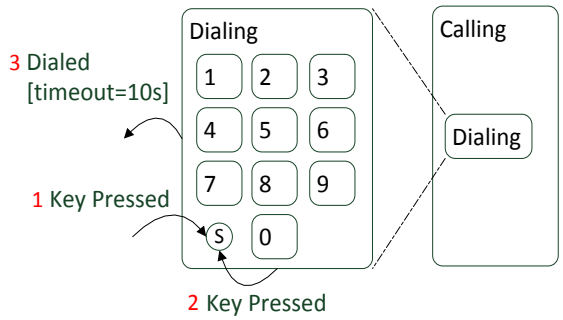


Figure 51. Unclustering the dial activity

Step 6: An empty arrow means that the incoming and outgoing interactions occur with distinct systems. This case was discussed previously in this section (see Figure 49) and results in Step 7.

Step 9: The PABX system receives the event *Answer* caused by the receiver system answering the call (Step 8) and forwards the event *Connected* (Step 9) to both the caller and the receiver systems (multicast) resulting in Step 10. This is why Step 10 is represented twice in the diagram. However, it does not mean that a system will receive the event *Connected* twice. The Calling and Receiving states are not orthogonal, thus the caller and receiver systems will receive the same event *Connected* in distinct scenarios. Because they are overlapping states and a Phone-*i* system can be either a caller or a receiver, we can model both roles in the same diagram. When applicable, this approach minimizes the number of diagrams necessary to represent alternative behaviors of a system.

Many scenarios with deviations can be easily extracted from the diagram of Figure 50. In the following we present one of them as an example.

1. Phone-*i* system is idle (default)
2. The handset is lifted to start a call.

17. A buzz is heard (no dial tone)

12. The handset is replaced to end the call

In the PCB-statechart of Figure 50, suppose that only some models of Phone-*i* enable the user to configure exclusive functionalities, like displaying the caller ID. As mentioned, every state of Phone-*i* inherits its *Broad* extension ○ by default. Thus, the new *Configuring* state should be an overlapping state of all Phone-*i* systems.

To tackle this problem the next modeling extension attaches the proposed symbolic notations to states (or generically to blobs (Harel, 1988)).

5.2.4. Statechart extensions applied to states

When a symbolic notation is attached to a state, the resulting notation overrides the default notation inherited from the parent state and enables the multiplicity of the child state to be represented individually. In the example of Figure 52, the symbolic notation *Multi* ⓜ is attached to the new state *Configuring* to limit its scope to ‘some models’. This overrides the default notation inherited from Phone-*i*.

Figure 52 abstracts the PCB-statechart of Figure 50. Phone-*i* ○ represents all phone systems. Because of the inheritance, the states *Operating*, *Calling*, and *Receiving* are

overlapping states present in all those phones. The state *Operating* encapsulates and abstracts the states *Calling* and *Receiving* of Figure 50.

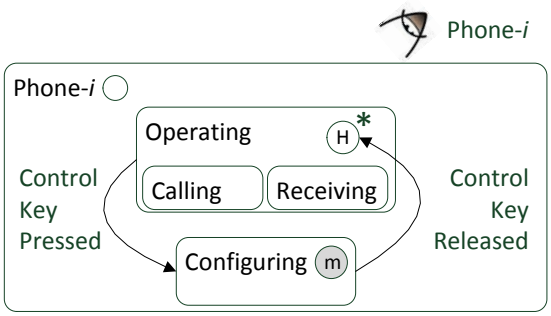


Figure 52. Example of symbolic notations attached to states

When a Control key is pressed in any of these states it triggers a transition to the state *Configuring* ⓜ only if this function is available in Phone-*i*. The notation *Multi* ⓜ overrides the notation *Broad* ○ of Phone-*i* and restricts the configuration capability to 'some systems'.

When a Control Key is released it triggers a transition back to the last active state because of the H* entrance of the state *Operating*. In the real world, this model could represent, for example, a user increasing or decreasing the volume of the handset. Moreover, it could be an abstraction of complex scenarios like those modeled by Harel (1987) to represent the configuration of a multi-alarm watch.

It is important to notice that an assertion can be assigned to the *Multi* notation ⓜ of the Configuring state exactly as discussed in Section 5.2.2 and illustrated in Figure 46. For example, an OCL assertion like Phone-*i*->isExtendedModel() can be assigned to the *Multi* notation of Figure 52 to define clearly in which subset of Phone-*i* the *Configuring* state overlaps. In this case, the assertion restricts the subset of overlapping systems to those phones that have extended characteristics that can be configured.

5.3. Modeling systems' interactions

In this section we exemplify how the proposed statechart extensions can be used to model systems' interactions. The given example comprises Skype™-like systems that extend the Phone-*i* systems of Figure 50 by allowing, for example, conference calls and chats. The use case diagram of Figure 53 shows the functionalities that we are going to discuss in more detail. It is important to note that we have considered only a small subset of Skype™-like system functionalities to illustrate our approach. Many events were omitted to keep the example simple.

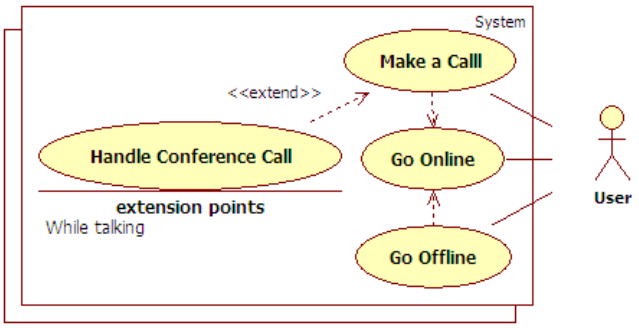


Figure 53. Skype™-like system use cases

Different from eliciting requirements of traditional systems, which run standalone, react to stimuli from bounded environments, and commonly do not affect other systems' behaviors, communicating systems like Skype™ need to be defined additionally in terms of interactions. In fact, each use case of Figure 53 comprises at least one interaction with another system that affects somehow the behavior of both systems.

Because the user must be online to interact, the use case Go Online must precede all the others. In the diagram, this is represented by the dependency relationships connecting *Make a Call* and *Go Offline* to *Go Online*. The extension point defines that *Handle Conference Call* is performed whenever a user makes a call to another user other than the ones that he/she is currently talking to.

When a user goes online, his/her contact list is updated with the most recent Online Status of each contact of the list. Similarly, the contact lists of his/her contacts that are currently available online are updated with the actual user's Online Status. Thus, the post-condition of the *Go Online* use case would be: "Online Status updated both locally and remotely". It is important to notice that this condition refers clearly to the internal and external effects of the use case.

When a user calls a contact currently available online in his/her contact list, the receiver's system will ring. At this moment the receiver can decide to accept, deny or simply to not answer the call. If the call is accepted, the users can start talking. Then, other users can be similarly added to the call anytime by any participant to handle a conference call. The call ends when all participants have left it either by hanging up or going offline.

Even though we can create use cases to describe these scenarios methodically, the resulting textual description of the system may not be the best option to support high level discussions about it during the initial phases of the project. The text might grow fast and

become more complex as the number of interactions grows. Moreover, changes involving these interactions may require great effort to maintain the use cases.

In this context, building a visual representation of the system and its interactions, yet with some formalism, could be a useful complementary modeling approach. It allows visual elements to be added to or removed from the model easily while the system is being defined and refined. Finally, the resulting PCB-statechart can support engineers to go deep into the details using appropriate resources.

When modeling system's interactions, engineers usually direct their efforts to the modeling of collective behaviors based on roles and rules, e.g., Kotov (1997). Roles mostly describe particular capabilities of the participant systems and how they can contribute to achieve more comprehensive goals. Thus, modeling roles often means describing only relevant capabilities of each system and yet in a detail level enough to enable the understanding of their characteristics. On the other hand, rules usually describe interaction mechanisms that every participant system must comply with to support a stable collective behavior. Thus, modeling rules frequently means describing in details when, where, how, and even for how long, the interactions must occur, as well as what systems are involved.

The PCB-statechart of Figure 54 represents visually the behaviors described previously for interacting instances of a Skype™-like system. The following variables are used to handle information and create more complex behaviors like in extended state diagrams.

#On = the number of contacts available online;

#Tk = the number of users currently talking; and

bOff = the initial Online Status (TRUE = Offline).

The scope of these variables is bounded by the system, i.e., each system has its own set of variables. Certain functions are invoked during transitions or interactions in Figure 54. Table 6 has a description of these functions.

Table 6. Description of Skype™-like functions

Function	Description
update	Update user's contact list according to recent online and offline events
notify	Notify user about a contact becoming online
set	Synchronize the list of participants in a conference call
getOnlineUsers	Get the list of users currently online

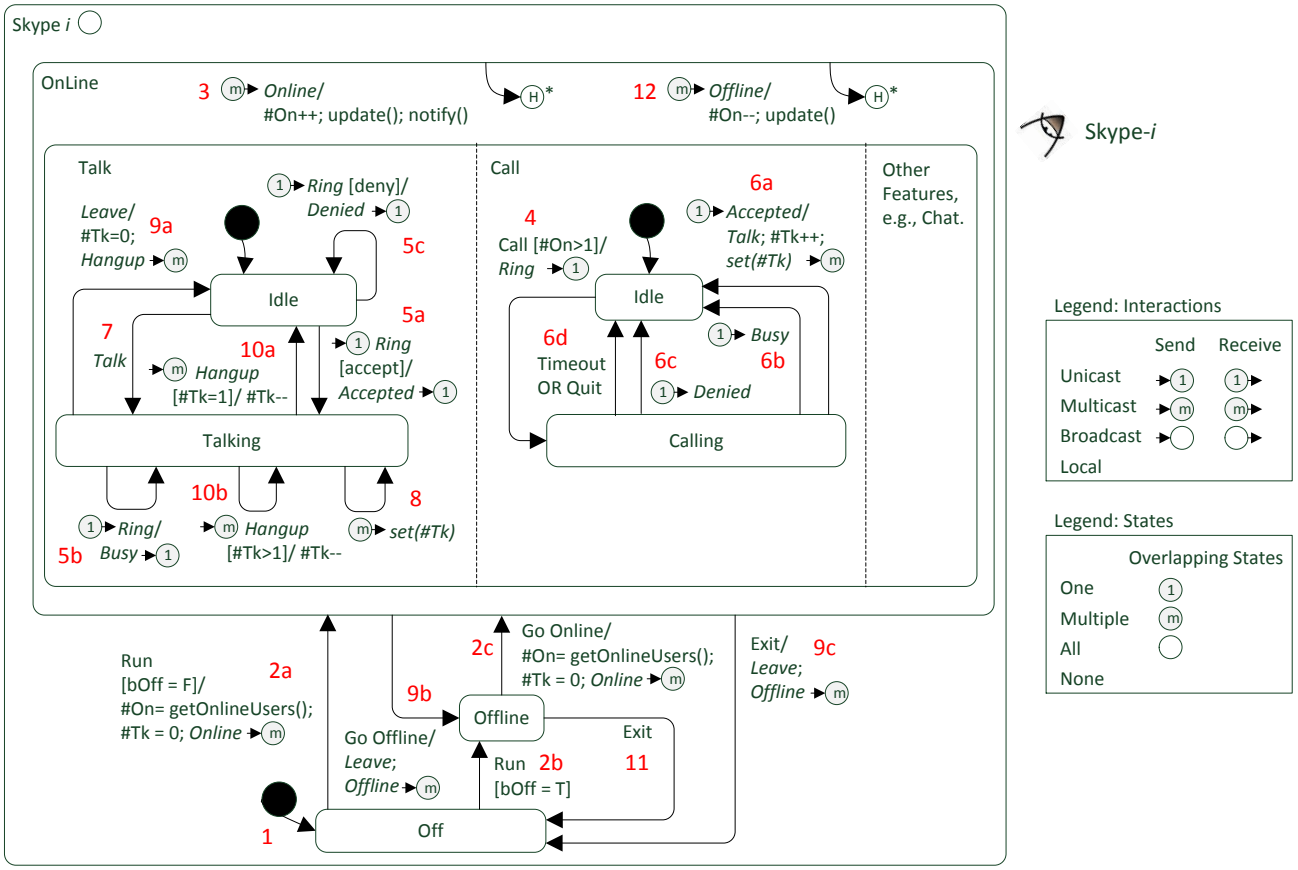


Figure 54. PCB-statechart of a Skype™-like system

Skype-i ○ denotes that all systems are similar and that all states overlap by default. All the interactions are of type *Uni* ① or *Multi* ⓜ because the number of systems is limited by the number of contacts in the contact list. Because all the states overlap, we can point to seemingly ambiguous scenarios in the diagram. For example, when a system is in the state configuration (*Talk.Idle*, *Call.Idle*, …) and the user makes a call from *Talk.Idle* (4), one may think that the resulting *Ring* event (4) would be broadcast internally and trigger transitions in orthogonal states, like *Talk.Idle* (5a, 5b, or 5c). That would represent an ambiguous scenario where the user could be either calling himself or another user. The proposed statechart extensions make such situation unambiguous.

The outgoing *Unicast* extension ① attached to the *Ring* event (4) represents that the interaction occurs uniquely and directly with another system, that will receive the *Ring* event exclusively. In this scenario, the notations 5a, 5b, and 5c relates to events of an overlapping *Talk* state in the receiver system. It is important to notice that a Skype-*i* system can be either a caller or a receiver. This is why the events *Call* (4) and *Ring* (5a, 5b and 5c) coexist in the same diagram.

On the other hand, when a user (caller) is going to start talking (6a) the *Talk* event is broadcast to orthogonal states exclusively and will trigger a transition from *Talk.Idle* to *Talk.Talking* (7) in the caller system.

When the PCB-statechart of Figure 54 is checked thoroughly by the stakeholders and they agree that it is correct, i.e., that it describes visually the expected behavior of a Skype-*i* system, software engineers can go further in describing, modeling, and designing the system. During these processes, the PCB-statechart can be used to support traditional SE approaches.

PCB-statecharts can be used since the very beginning of the development processes. Thus, they are different from those UML diagrams that are used later because they comprise specialized elements such as classes and objects that are not available initially. Using PCB-statecharts, interactions can be visually represented even when systems are simply abstract rounded rectangles (blobs) like those of Figure 47. This way, they can support different SE approaches from requirements to development. Moreover, they can be used, for example, as a reference to textual descriptions like use cases, as exemplified in Figure 55.

Requirements engineers can embed in their textual descriptions references to the visual elements of a PCB-statechart that represent the same behavior being described. This approach can help, for example, to prevent ambiguities when describing systems' interactions.

In Figure 55, the numeric identifiers of the PCB-statechart of Figure 54 are used to reference specific behaviors in that diagram (see the Main successful scenario). They are embedded in the textual description that focuses mainly the behaviors that comprise at least one interaction among systems. Intrinsic behaviors are omitted intentionally and only successful scenarios are described. The use case *Handle Conference Call* extends the use case of Figure 55 whenever the caller *Makes a call* while he/she is currently talking (Precondition). If succeeded, the receiver is added to a conference call (Post condition).

When a participant leaves the conversation (9a), goes offline (9b), or leaves the system (9c) all the remaining participants are notified because the *Leave* event causes the *Hangup* event to be sent in multicast. If there are two or more participants still talking (#Tk > 1), the *Hangup* event just decrements their #Tk (10b) and they can keep talking (*Talk.Talking* state active). Otherwise (#Tk = 1), the conversation ends (10a) and the last participant returns to the *Talk.Idle* state. In all those scenarios, the participant leaving the conversation has his #Tk counter set to zero and the *Talk.Idle* state becomes active.

UC - Make a Call	
Scope: Skype™-like system **Level**: User goal **Primary actor**: Skype-*i* user (caller) **Stakeholders and Interests**: - Skype-*i* users (Caller and Receiver): . Want to be able to talk to each other **Preconditions**: - Caller and Receiver are online. - Caller is currently not talking. **Postconditions**: Caller and Receiver talk to each other **References**: PCB-statechart of Figure 54	
Main successful scenario	
1	Caller: Calls a contact of his/her contact list. (4)
2	Receiver: Accepts the call. (5a) The caller is notified immediately.
3	Caller: Prepares for talking (6a). Updates the number of people talking
4	Caller: Start talking (7).
5	Receiver: Updates the number of people talking (8)
6	Caller or Receiver: Leaves the conversation. (9a) The partner system is notified immediately.
7	Caller or Receiver: Hangs up. (10a)

Figure 55. Use case: *Make a Call*

The PCB-statechart of Figure 54, in the abstraction level it was modeled, can provide information to help engineers to design appropriate architectures. For example, *Talk* and *Chat* are orthogonal states that represent different functions of the system. If they are intended to be supported by the same network services, the "Layers" architecture pattern (Buschmann et al, 1996) may be candidate architecture for this system, as shown in Figure 56.

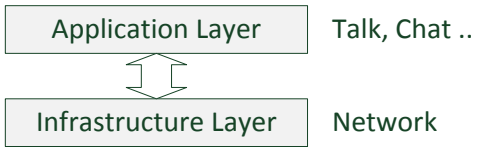


Figure 56. Layered architecture

Here, we are just discussing about how PCB-statecharts can support different software engineering areas. It is not our intention to argue about the best options to develop a Skype™-like system.

The PCB-statechart of Figure 54 is also full of details to support the development phase. Variable #Tk, i.e., the counter for users currently talking, is incremented in (6a) and synchronized in (8). But, it does not mean that #Tk is updated twice in a system when a call is made. Actually, it is updated just once in each system participating of the call. This is because the related events, i.e., *Accepted* and *Set*, occur in different systems. While *Accepted* occurs in the caller's system (6a), *Set* occurs in all other systems (Multicast) currently talking (8). Thus, every time a new user enters the call (or conference call) all the participant systems update their own counter #Tk. The same happens when a system leaves the call (9a). A *Hangup* event (Multicast) will cause the decrement of the counter #Tk in every remaining system of the call (10b). However, if #Tk=1, the call ends and the last system returns to the *Talk.Idle* state (10a). A similar synchronism is managed by the variable #On, i.e., the counter for contacts currently online. When a user goes online (2a or 2c), offline (9b), or exits the system (9c) all contacts currently online in his/her contact list are notified (Multicast) of the new Online Status (3 or 12).

Finally, one may say that the PCB-statechart of Figure 54 does not reflect accurately the behavior of a Skype™-like system. For example, that a server receives events and then distributes them appropriately to the systems. Thus, there are not direct multicast interactions among systems like in (2a, 2c, 9a, 9b, and 9c). However, the example was intentionally designed this way to direct the attention to the flexibility of PCB-statecharts to support changes. If such server really exists, or it should exist to improve the system architecture, it could be added easily to the diagram just like the PABX system of Figure 50. Then, source-to-targets multicast interactions (1→n) should be changed by a pair source-to-server unicast interaction (1→1) and server-to-targets multicast interaction (1→n).

5.4. Related work

To the best of our knowledge, PCB-statecharts is the first proposal to the modeling of systems' interactions that comprise all the following characteristics: a) separation of concerns by considering different viewpoints; b) symbolic notation to represent interactions that eliminates the need of connecting related entities in the model; and c) notions of multiplicity of systems, and concurrency and parallelism among systems.

In this section we present related works that fulfill at least one of these characteristics to discuss improvements and constraints. Actually, none of them really motivated this thesis. However, they are good examples of the variety of domains in which PCB-statecharts would

fit suitably. They include SoS and real-time distributed control systems as discussed in the following.

Kotov (1997) proposed the modeling of SoS by means of a hierarchical and concurrent structure that represents the SoS components and the communication between them. His main goal was to view large information systems in a uniform and systematic way as Communicating Structures. The main modeling activities are related to the coordination of data traffic and data placement. Figure 57 exemplifies the modeling of SoS as a Communicating Structure (Kotov, 1997).

The system's components are represented simply as nodes. Each node has memory that may contain items. Nets are groups of links that connect nodes. The items are generated at some nodes and move from node to node along links, with some delay. The item traffic models the message and data traffic in systems as in Figure 18.

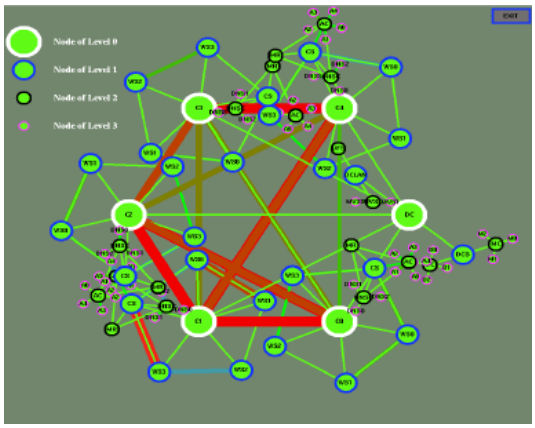


Figure 57. Modeling SoS as a Communicating Structure (Kotov, 1997)

Kotov (1997) argues that SoS models should be as simple as possible without losing those features that are important for the system validation. He states that modeling methods based on statecharts often do not meet these characteristics because they are biased to support rigorous and efficient designs and development processes.

We think that PCB-statecharts can bridge the gap between simplicity and rigorousness as discussed previously. However, because of the reactive nature of the systems modeled by statecharts, interactions are event-based rather than information-based relationships. Thus, extra efforts should be made toward the analysis, specification, and validation of information moved along with interactions in PCB-statecharts.

Using statecharts to support requirements specification was also proposed by Glinz (2002). He explored how a statechart variant for requirements models should look to be as simple as possible, easy to understand and well suited for expressing requirements. Three

characteristics were considered essential: 1) typical behavioral and interaction requirements must be expressible with reasonable effort; 2) statechart models, data models and functionality models must smoothly fit together; and 3) state and state transition explosion must be avoided.

Truthfully, all statechart variants proposed by Glinz (2002) could extend PCB-statecharts as well. However, there is one important difference. From one system to another, events have to be transmitted explicitly via channels (Leveson et al, 1994) that must provide the information where the event comes from. Figure 58 shows Glinz (2002)'s integrated object/statechart diagram that provides the requirements for a Room Control of a heating control system.

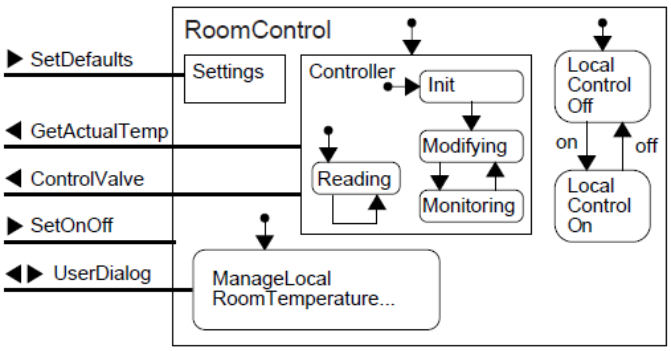


Figure 58. Requirements for a Room Control [13]

The symbolic notations we proposed to statecharts can bring several benefits to the understanding of the presented requirements. Mostly, they are more precise than channels to identify interaction points. For example, the point where SetOnOff produces an effect in the room control cannot be affirmed unless intuitively. Thus, if the number of channels grows, the understanding of the model can decrease fast. Just like arrows connecting systems, if all lines are drawn to connect channels to their end points in complex systems, the model may become a puzzle.

Tian et al (2011) proposed a method to model visually interaction relationships among entities. It comprises the following steps: 1) define the internal logic of each entity related to the interaction relationship in entity containers; 2) define the interface of each entity in the interaction space; and 3) define the interaction logics among interfaces in the interaction space. Figure 59 shows how the resulting model looks like.

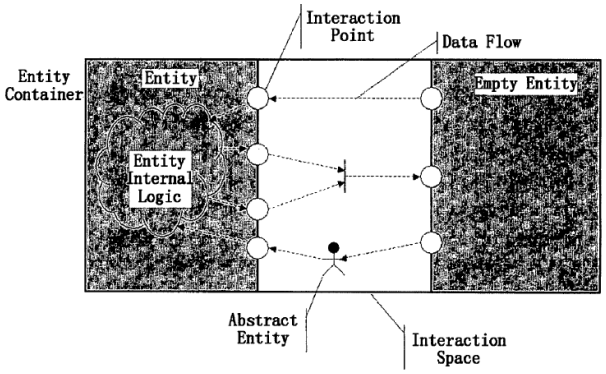


Figure 59. Modeling relationships among entities (Tian et al, 2011)

The problem related to a growing number of systems that interact may also apply to the presented method. For example, it would be hard to create a model with several systems interacting among them concurrently as usual in SoS environments. Similarly to the approaches discussed previously, complex relationships can cause an explosion of lines connecting interaction points across the interaction space.

5.5. Case study – Modeling the sample virtual SoS

The sample SoS presented is this section was introduced in Section 4.4, where use cases and UML diagrams were used to describe the SoS requirements in terms of scenes and their related scenarios. However, we discussed the problem of using this approach, which may lead to complex use cases descriptions and confusing diagrams as the number of systems and interactions grows (see Section 4.5). In this case study, we improve the SoS requirements of Section 4.4 with PCB-statecharts to solve this problem. We show how interactions described in terms of use cases can be visually represented to improve the understanding of the SoS dynamism.

To exemplify the evolutionary property of SoS, we assigned concrete deliverables to the publisher and processor roles of the sample SoS described in Section 4.4, which are location and video respectively. The SoS feature model of Figure 33 was updated to represent this change as shown in Figure 60.

The fact of defining the location and video deliverables to the sample SoS revealed unexpectedly new SoS characteristics. In Section 4.4.1 we presented this SoS as being virtual because it lacks a central management authority, a coordinator and a centrally agreed mission, and presents emergent behaviors (see SoS classification in Section 2.2.3). However, this classification can change by simply assigning a mission to the SoS, for example: "The

Processor must record a video of the environment around it whenever the location informed by the Publisher points to coordinates inside a configured bounded area".

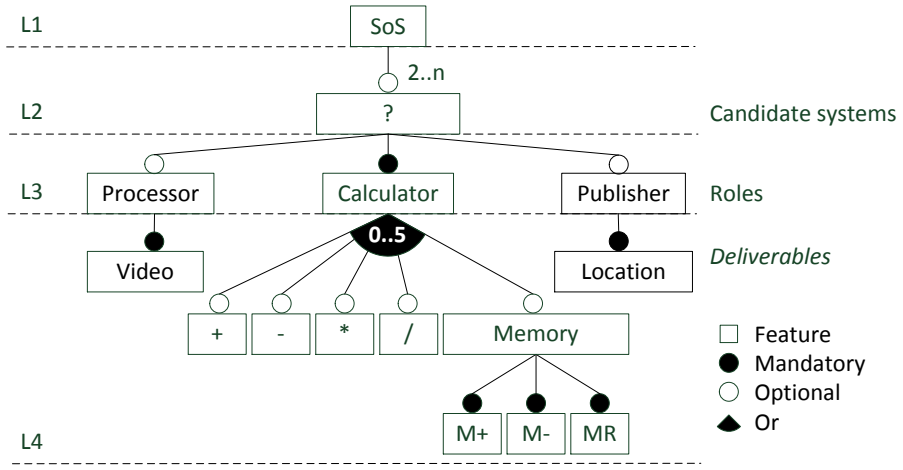


Figure 60. Sample SoS feature model updated with location and video deliverables

In the above example, the SoS can be *collaborative* because central players, i.e., Publisher and Processor, interact to fulfill agreed upon central purposes. Also, it can be *directed* because the normal operational mode of these systems is subordinated to a central managed purpose, i.e., the mission. Finally, it can be *virtual* because the calculators need no coordination. To the best of our knowledge, this scenario has never been discussed in the literature. We think it will become more evident as the number of systems grows. Actually, different missions can be assigned to distinct sets of SoS members rather than to the whole SoS. In this case, the SoS cannot be uniquely classified as proposed, for example, by Maier (1998) and Jamshidi (2009).

It is important to notice that the behavior described by the above mission could also emerge naturally in the sample SoS when location and video deliverables are made available. In this case, the initial SoS \classification, i.e., *virtual*, would not change.

Figure 61 illustrates the PCB-statechart we modeled to represent visually the sample SoS. L and R prefixes were used to distinguish local and remote activities respectively as in L-Add and R-Do. Next, we discuss the details.

The *Calculator*, *Processor*, and *Publisher* roles of Figure 61 run concurrently in separated processes. The Calculator comprises operation, calculation and discovery services, i.e., Composition, Parse+, and Parse- that run in distinct threads. This way, the system can process interactions concurrently with its regular assignments. The Publisher comprises publish and subscribe services that run in distinct threads as well. This way, the system can publish information and receive information published by other systems concurrently. When

the system is running out of the SoS, i.e., autonomously, the operation thread keeps running to perform the system's regular assignments.

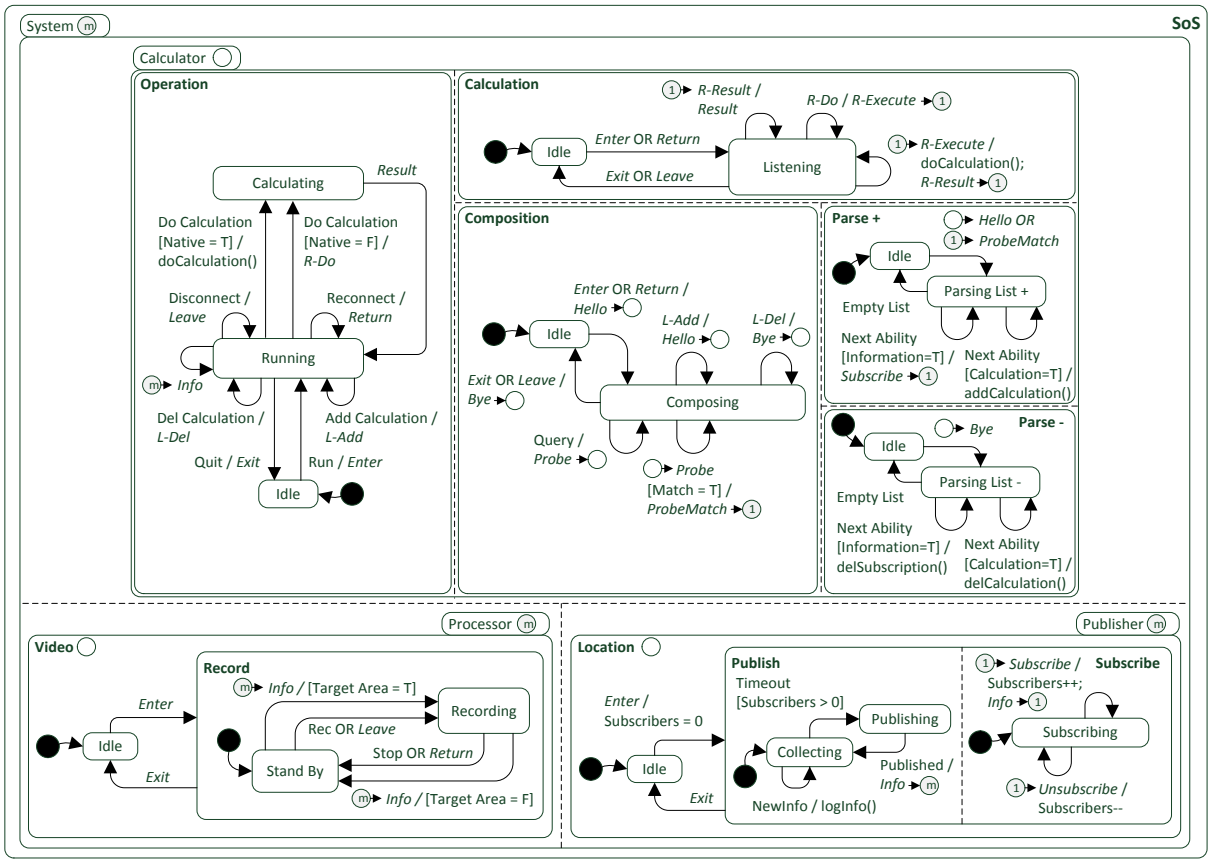


Figure 61. Sample SoS PCB-Statechart

According to Table 4, System (m) represents a finite number of systems that composes the SoS and overlaps because of the mandatory role Calculator ◯, played by all systems, and the optional roles Publisher (m) and Processor (m), possibly played by some but not necessarily all systems (Figure 60). These (orthogonal) roles are played concurrently by the systems. By inheritance, all Calculators' internal states overlap. Differently, the *Broad* notation assigned to Video ◯ and Location ◯ explicitly overrides the *Multi* notation (m) assigned to their parents, i.e., Processor and Publisher respectively.

Calculator ◯ comprises the minimal functionality of a member system. Internally, Operation ◯ describes the system running both independently and cooperatively. In the first case, considering the system is not running, the active state changes from *Operation.Idle* to *Operation.Running* when the system runs and vice-versa when it quits. *Operation.Calculating* is reached from *Operation.Running* whenever a calculation is requested. The inverse occurs

when the calculation is finished and a result is available. In the SoS scenario, this flow is enriched with broadcasts between orthogonal states and interactions among systems.

When the system initially runs to compose the SoS, its active state changes from *Operation.Idle* to *Operation.Running* and the event *Enter* is broadcast to all orthogonal states. Then, a Composition◯ starts and a *Hello*◯ event is sent to all SoS members (*broadcast*) to announce the new system and inform the abilities it can share. The *Composition.Composing* state is reached and it will manage continuously and concurrently all further composition activities while the system belongs to the SoS.

Whenever a *Hello*◯ event reaches the *Parse+.Idle* state of an active SoS member (receiver), the *Parse+.ParsingList+* state is activated and it will parse the list of abilities shared by the sender. Every calculation in the list that is not available locally is registered (calculation + sender's endpoint) and becomes available to the receiver (addCalculation( )). Also, for every information in the list that is useful for the receiver, a Subscribe① event (subscription request) is sent *unicast* back to the sender, i.e., directly and exclusively to it. When the list becomes empty, i.e., the parse has ended, the *Parse+.Idle* state becomes active again.

After announcing itself and informing its abilities to the whole SoS, the new system advances the composition process by querying all SoS members for their abilities. For this purpose, it sends a *Probe*◯ event while in the *Composition.Composing* state. Whenever this event reaches the *Composition.Composing* state of an active SoS member (receiver), it will cause that system to match its list of native abilities with the list of abilities being probed. If the resulting list is not empty, a *ProbeMatch*① event with that list is sent *unicast* back to the sender. Finally, when *ProbeMatch*① events are received back from the whole SoS and they reach the *Parse+.Idle* state of the new system, the *Parse+.ParsingList+* state is activated and the parse process described above for SoS members is performed locally by the new system.

When the composition process ends, the new system becomes an SoS member and can benefit from all the abilities currently being shared. As the *Composition.Composing* state remains active, the system will welcome future systems joining the SoS exactly as described above.

While the system is running, i.e., *Operation.Running* state is active, it can decide to leave the SoS anytime either for a moment (*Disconnect*) or for a longer period of time (*Quit*). Before leaving, however, it must announce its decision to the whole SoS. For this purpose, it

broadcasts respectively a *Leave* or an *Exit* event to all orthogonal states. Then, a *Composition* ◯ ends and a *Bye*◯ event is sent to all SoS members (*broadcast*) to announce the system and inform the abilities just becoming unavailable. The *Composition.Idle* state becomes active and no more compositions are handled by the system until it decides to enter the SoS again by either broadcasting a *Run* or *Return* event, depending on the way it left the SoS previously.

Whenever a *Bye*◯ event reaches the *Parse-.Idle* state of an active SoS member (receiver), the *Parse-.ParsingList-* state is activated and it will parse the list of abilities shared by the sender. Every calculation in the list that was previously registered by the receiver is unregistered (delCalculation( )) and becomes unavailable to it. Also, remove subscriptions (delSubscription( )) for every information in the list to which the receiver has an active subscription. When the list becomes empty, i.e., the parse has ended, the *Parse-.Idle* state becomes active again.

When a shared calculation is requested in the *Operation.Running* state, an *R-Do* event is broadcasted to all orthogonal states. Then, the *Operation.Calculating* state becomes active and the system waits for the response. If the *Calculation.Listening* state is active, it will send *unicast* an *R-Execute*① event to the registered endpoint associated with the requested calculation. If the *Calculation.Listening* state is also active in the target system, it will handle the *R-Execute*① event, extract the calculation parameters, perform the calculation and return an *R-Result*① event *unicast* to the requester. Then, in the requester, the *Calculation.Listening* state handles the *R-Result*① event, extracts the result value, and *broadcasts* a *Result* event to all orthogonal states. Finally, the *Operation.Calculating* state handles the *Result* event and the result value. The *Operation.Running* state becomes active and the system is ready for a new request. When the system is requested to perform a shared calculation, the process behaves similarly.

Publish.Collecting and *Subscribe.Subscribing* states become active simultaneously when the system enters the SoS and the *Enter* event is broadcast to all orthogonal states. Whenever updated information becomes available it is logged for publishing (logInfo() in *Publish.Collecting*). When the time interval between publications times out and there are subscribers, the *Publish.Publishing* state becomes active and the information is published through an event Info ⓜ, which is sent to all subscribers. Then, the *Publish.Collecting* state becomes active again and the publishing cycle restarts. To support the SoS mission described above, the *Info* event must contain updated location information.

The *Subscribe.Subscribing* state, when active, listens continuously to Subscribe① and Unsubscribe① events to keep the list of subscribers up to date. These events result from the parsing activities performed in the *Parse+.ParsingList+* and *Parse-.ParsingList-* states respectively of other SoS members as described previously.

The publish/subscribe activities stop, i.e., *Location.Idle* state becomes active, when the system quits the SoS and the *Exit* event is broadcasted from the *Operation.Running* state to all orthogonal states. It is important to notice that the *Publish.Collecting* state remains active when the system disconnects momentarily from the SoS. In this case, updated information keeps being logged continuously but no publish/subscribe activities are performed until the system reconnects to the SoS and the lists are filled again.

When the system enters the SoS and the *Enter* event is broadcast to all orthogonal states, the *Record.StandBy* state becomes active. If the system is engaged in the SoS mission, it listens continuously to Info ⓜ events published in the SoS. Whenever this event contains a location that points to coordinates inside the configured bounded area, the *Record.Recording* state becomes active and the system starts recording a video of the environment around it. It will perform this activity until an Info ⓜ event indicates it has left the bounded area. In this case, the *Record.StandBy* state becomes active again and the recording cycle restarts.

If the *Record.StandBy* state is active and either the system disconnects momentarily from the SoS or a *Rec* event is broadcasted, the *Record.Recording* state becomes active and the system starts recording a video immediately. Later, if either the system reconnects to the SoS or a *Stop* event is broadcasted, the *Record.StandBy* state becomes active again and the recording activity ends. The origin of the *Rec* and *Stop* events are not represented explicitly in the PCB-Statechart of Figure 61. They could be part of a contingency mechanism that, for example, can decide to start capturing video whenever all publishers become unavailable.

5.6. Final considerations

Despite our particular interest in modeling SoS, we think that the proposed statechart extensions can also be useful to model other types of systems like those in which interactions among networked systems need to be properly designed before they can communicate. Examples include distributed systems and service-oriented systems (Lewis et al, 2001).

Using observers to support the modeling of interactions can be useful in the SoS context. Actually, because of the managerial independence of the systems, SoS engineers may be required to deliver requirements specifically to each candidate system. In this case, a set of

different but related PCB-statecharts can be designed together to fully describe all the interactions and, then, be delivered specifically to different teams of system engineers.

Our ongoing work includes using PCB-statecharts to model SoS of conventional systems like smartphones, retail devices and others. We want to expand the original concept of SoS that defines its members as large-scale, often complex, and information-based systems (Maier, 1998). We believe that any system can contribute somehow with abilities necessary to solve more complex tasks. Thus, PCB-statecharts aim to support our goal of putting systems to work together and make them to collaborate to produce useful and more complex results. For example, building unmanned vehicles as SoS.

The proposed SoSE approach (Figure 23) intentionally does not include a process for suppliers so that they can deliver SoS members. Ideally, SoS engineers should not suppose anything about the development of the member systems. Instead, they should provide only detailed information about interactions as we did up to this point. However, to achieve our objective of building the sample virtual SoS, it is necessary to go further and provide a way to develop also the member systems. Next, we propose extensions to the general process for SPLE presented by Ziadi, Jézéquel, and Fondement (2003) (Figure 12). Then, we show how these extensions can support the development of SoS members.

Chapter 6

An SPLE extension towards SoS development

6.1. Initial considerations

In the SoS context, SPL can play the role of deploying useful SoS members, probably in a more flexible way than other development paradigms. In fact, the activity of instantiating different products from SPL to meet distinct SoS requirements tends to be more effective than maintaining a group of systems repetitively. However, not all systems are SPL instances, thus we are aware that conventional systems are equally important to compose SoS. But, they are not our focus in this chapter.

Because SPLE can deliver families of products and SoSE aims to compose systems to achieve unique goals, we are wrongly induced to think that they relate directly to each other, i.e., SPLE deploys families of systems and SoSE builds SoS by composing these systems. However, SPLE has not been evolved toward the design of interoperable systems capable of contributing to SoS goals (see Weiss and Lai (1999), Atkinson et al. (2001), Gomaa (2004), Sochos, Riebisch, and Philippow (2006), Lee and Kotonya (2010) and others). In fact, to the best of our knowledge, this is the first work that attempts to integrate SPLE and SoSE processes.

We are motivated by the possibility of bridging the gap between SPLE and SoSE so that they can contribute mutually. Based on the SPL characteristics that do not direct to the development of SoS members, e.g., determinism x opportunism (see Sections 2.3.2 and 2.3.4), we propose SPLE extensions to enable the design of families of systems able to compose SoS more naturally. For that, we do not impose constraints to the SoS constituent systems, which may exist or not, be simple or complex, and have the same or different origins. Indeed, we want to be able to compose virtually any system capable of contributing somehow to the SoS goal.

This chapter is organized as follows. In Section 6.2, we propose an SPLE extension towards SoS development. The objective is to be able to deploy SoS members with minimal impact on the regular SPL productivity. In Section 6.3, we evolve the case study of Sections

4.4 and 5.5 by deploying members to the virtual SoS. Finally, in Section 6.4, we present our final considerations.

6.2. The proposed SPLE extention

The SoSE approach we proposed in this thesis (see Section 3.2) assigns the role of supplier to entities or processes able to deploy SoS candidate systems whenever the available systems do not fully meet the SoS requirements (see Figure 23). In this section, we propose extensions to SPLE so that the resulting approach can fit naturally the role of supplier in our SoSE process. The SPLE extensions are based on the following statements:

1. According to Figure 13, the earlier the 'break even' point is reached the greater will be the productivity and profitability of the SPL. However, this is only possible if the core assets available can fully support the building of the whole family of products with minimal maintenance (high reuse rate). Ideally, the SPL core assets should not be maintained to meet FoS requirements, as this would decrease the SPL profitability. However, this is very likely to occur because FoS requirements are supposed to change over time to comply with the SoS dynamism. Thus, SPLE processes needs to deal with those frequent changes and still keep the SPL profitable.

2. Systems' abilities should be designed in such a way they can be shared among SoS members through different mechanisms, e.g., agents and services. For that, they must be highly cohesive and weakly coupled to the SPL core assets so that they can evolve freely without affecting the regular SPL production.

Considering these statements, we propose the creation of FoS core assets apart from the SPL core assets specifically to support FoS requirements. SPL and FoS core assets relate closely to each other, although the first one is not supposed to have dependencies on the second one. Figure 62 shows the proposed SPLE extensions that enable SPLE and SoSE to work complementarily.

Notice that Figure 62 extends the general process for SPLE (Ziadi, Jézéquel, and Fondement, 2003) of Figure 12 by adding new activities at the right side. Also, it links to the proposed generic SoSE process of Figure 23 by playing the role of supplier. This link is represented at top right of Figure 62.

The Domain Engineering was extended by the Domain Extension activity which is started on demand whenever new FoS Requirements arrive at SPL production time, e.g., when SPL instances have to compose a new SoS. This activity is responsible for designing abilities

and components necessary to meet the FoS requirements, for example, read GPS coordinates (ability) and publish them through appropriate interaction mechanisms (components) to other SoS members. The resulting artifacts are stored in the FoS repository and can be reused more opportunistically.

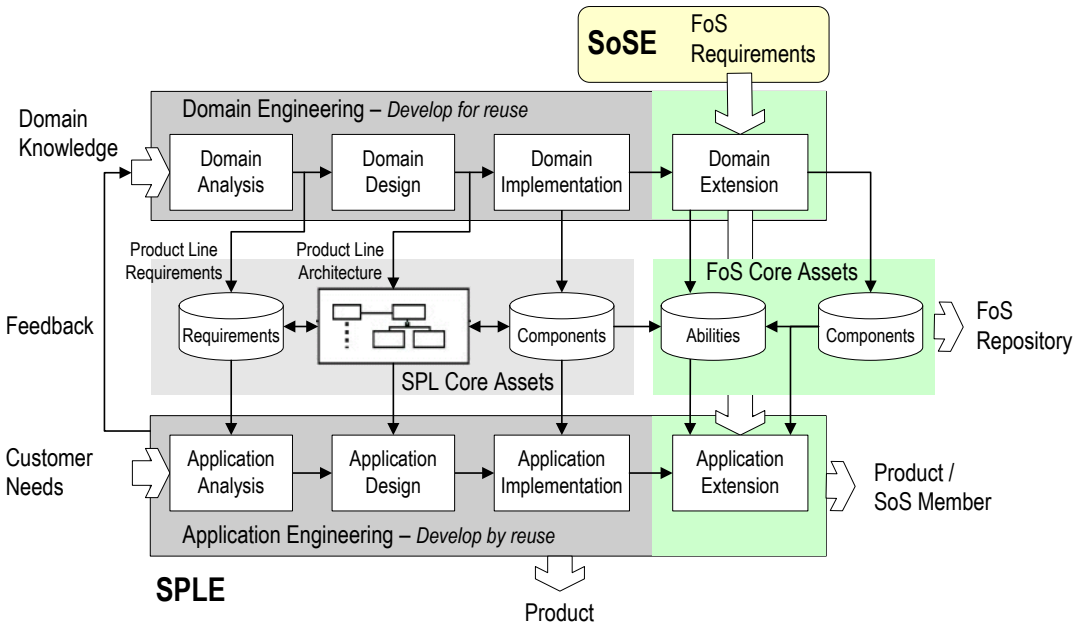


Figure 62. SPLE extensions

According to the above example, components and abilities refer to different artifacts with different purposes in the FoS repository. Components are intended mainly to enable SPL instances to compose different SoS and interact with their members, for example, there may be components to enable SPL instances to provide and consume services in service-oriented SoS according to the collaboration rules defined by the SoS engineers. On the other hand, abilities refer to native capabilities of each SPL instance that can be shared with other SoS members, for example, read GPS coordinates. Thus, components basically support the sharing of abilities among SoS members.

A many-to-many relationship exists between abilities and the interaction mechanisms provided by the components of the FoS core assets. Indeed, an interaction mechanism, e.g., services, can be used to share several different abilities, e.g., information and calculation. Additionally, an ability can be shared through several different interaction mechanisms. This independence is the reason why abilities and components appear separately in the SPLE approach of Figure 62.

The Application Engineering was extended by the Application Extension activity of Figure 62, which starts on demand whenever a new SoS member must be built and deployed

at SPL production time. This activity links SPL instances to components and abilities of the FoS core assets to instantiate SoS members that meet the FoS requirements.

The SPLE extensions that we proposed may apply in different ways to distinct SPL approaches. Exploring all possibilities is out of the scope of this thesis. However, we think that the given extensions can support the evolution of SPL products to SoS members in many existing scenarios, e.g., DSPL.

6.3. Case study – Composing the sample virtual SoS with SPL instances

This case study was performed to state that SPLE can integrate SoSE processes to deploy SoS members (Ramos et al., 2013). It aims to fulfill the "Suppliers" phase of the SoSE process described in Section 3.2 (Figure 23). The extended SPLE approach of Figure 62 was applied to support the development of candidate systems for the sample SoS described in Sections 4.4 and 5.5. Only one SPL was intentionally developed to keep the case study simple but yet elucidative. However, instances from different SPL could also be used with minimal impact on the SoSE process.

For our benefit, the chosen scenario revealed an SoS composition strategy involving different instances of the same SPL, which interact to share distinct abilities and to improve themselves collectively. This is an example of SoSE and SPLE working complementarily, which has not been explored in the literature. Actually, complex systems, considered by most of the SoS approaches, usually do not belong to families of systems able to provide different instances of those systems, for example, to increase diversity and thus protecting the SoS against disruptive behaviors of member systems (see Table 1).

Although basically any system can perform calculations we opted for developing a basic calculator as an SPL because we can easily design its operations as optional features that can be enabled/disabled dynamically and conveniently. This is an important characteristic for us in our attempt to demonstrate the benefits of using SPL in the SoS context.

The requirements of the calculator as SoS member were described in the previous case studies (see Sections 4.4, and 5.5). Those requirements were extended to comprise specific characteristics, intrinsic to SPL. This activity is part of the Low-Level Requirements – Member Systems phase of the proposed scene-oriented RE approach described in Section 4.3 (Figure 28). The specific requirements that we added are:

1. Operations are optional features that can be activated dynamically in the systems' startup;

2. The activation of each operation must be independent and arbitrary. Memory functions (M+, M-, and MR) are the only exception and must be activated together as a group;

3. At least one feature must be active at runtime so that the system can run independently and still provide useful functionality.

The general SPLE process of Figure 12 guided the development of the calculator SPL according to the existing requirements. The SPL feature model used was that of Figure 60. Screenshots of two different SPL instances running as independent systems and the command lines used to start them are shown in Figure 63.

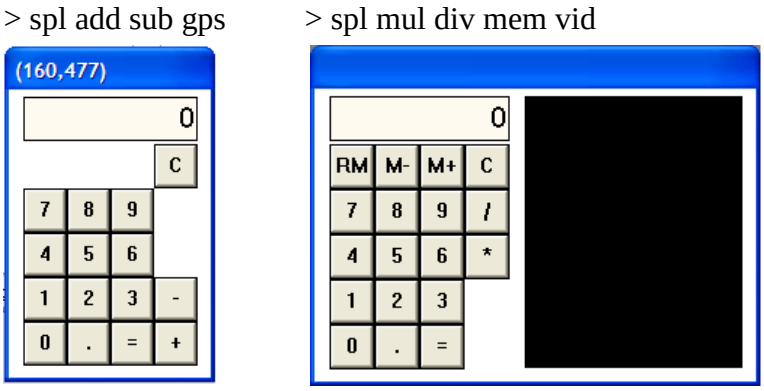


Figure 63. Two different SPL instances running as independent systems

The instance on left has Addition, Subtraction and Location (160,477 top left) features active. The instance on right has Division, Multiplication, Memory, and Video (black area) features active.

When the context changes to SoS, active abilities are shared among SoS members. For example, calculations can be requested on demand by a system to be executed by another system (see Sections 4.4, and 5.5 for interaction details). The most evident benefit is the fact that each member system will have available for its own purposes all the abilities shared in the SoS. For example, both calculators of Figure 63 will be able to execute all calculations shared in the SoS, i.e., +, –, *, /, M+, M–, and MR, even when they are not available natively. The result is shown in Figure 64, where the two calculators of Figure 63 now look like full calculators.

Next, we present details about the development we performed. The source code was written in ANSI-C using Win32 API.

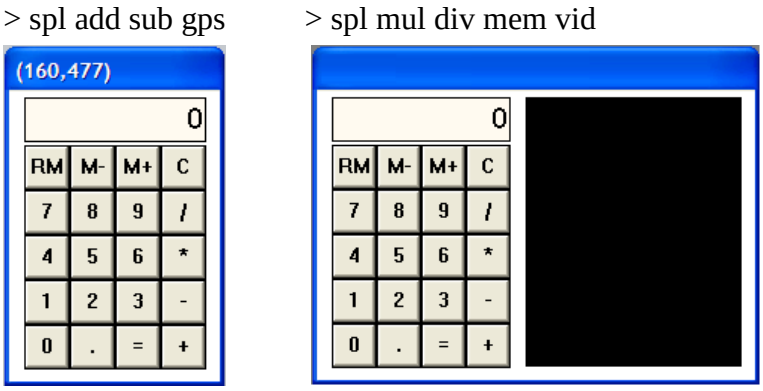


Figure 64. Two different SPL instances running as SoS members

Two bitmaps were created to manage local and shared (SoS) abilities respectively.

```
char    localOp = 0; // Local  abilities bitmap (char -> Max 8 abilities)
char   sharedOp = 0; // Shared abilities bitmap
```

Each ability was assigned to a particular bit of the bitmap.

```
#define       _NONE_  0x00
#define       _ADD_   0x01
#define       _SUB_   0x02
#define       _MUL_   0x04
#define       _DIV_   0x08
#define       _MEM_   0x10
#define       _GPS_   0x20
#define       _VID_   0x40
#define       _ALL_  (_ADD_ | _SUB_ | _MUL_ | _DIV_ | _MEM_ | _GPS_ | _VID_)
```

Later, this technique will enable discovery events to inform the availability of multiple abilities in a single bitmap. For example, when the calculators on the left and right side of Figure 63 enter the SoS, the *Hello* events informs `_ADD_|_SUB_|_GPS_` and `_MUL_|_DIV_|_MEM_|_VID_` respectively as shared abilities. Furthermore, bitwise functions can provide useful information quickly. See the examples below:

~ localOP (1's complement) : All abilities not available natively (must *Probe* the SoS)

~(localOP | sharedOp) : All abilities not available (cannot be executed/requested)

The availability of each ability can be verified by the following macros:

```
#define isAvailable_Local (op) ( localOp & op) // Available as native
#define isAvailable_Shared(op) (sharedOp & op) // Available as shared
#define isAvailable(op) \                      // Either native or shared
        isAvailable_Local (op) ||
        isAvailable_Shared(op)
```

Command line sets the native abilities available at startup (optional features):

```
if(*lpCmdLine)
{       localOp = _NONE_;

        // Enable native abilities
        if(_ftcsstr(lpCmdLine, TEXT("add")))    localOp |= _ADD_ ;
        if(_ftcsstr(lpCmdLine, TEXT("sub")))    localOp |= _SUB_ ;
        if(_ftcsstr(lpCmdLine, TEXT("mul")))    localOp |= _MUL_ ;
        if(_ftcsstr(lpCmdLine, TEXT("div")))    localOp |= _DIV_ ;
        if(_ftcsstr(lpCmdLine, TEXT("mem")))    localOp |= _MEM_ ;
        if(_ftcsstr(lpCmdLine, TEXT("gps")))    localOp |= _GPS_ ;
```

```
        if(_ftcsstr(lpCmdLine, TEXT("vid")))    localOp |= _VID_ ;
        if(_ftcsstr(lpCmdLine, TEXT("all")))    localOp |= _ALL_ ;

        if( localOp == _NONE_) // At least one feature must be available
            localOp |= _ADD_ ; // Default to addition
}
```

Abilities are either shown or hidden dynamically in the user interface according to their availability at any given instant (Figure 63).

```
ShowWindow(h1,isAvailable(_ADD_)? SW_SHOW:SW_HIDE); // Show or Hide the [+] button
ShowWindow(h2,isAvailable(_SUB_)? SW_SHOW:SW_HIDE);
ShowWindow(h3,isAvailable(_MUL_)? SW_SHOW:SW_HIDE);
ShowWindow(h4,isAvailable(_DIV_)? SW_SHOW:SW_HIDE);
ShowWindow(h5,isAvailable(_MEM_)? SW_SHOW:SW_HIDE);
ShowWindow(h6,isAvailable(_MEM_)? SW_SHOW:SW_HIDE);
ShowWindow(h7,isAvailable(_MEM_)? SW_SHOW:SW_HIDE);
// Show the video window if enabled
if(isAvailable_Local(_VID_)) // Video is a local ability
{       if(hCam)
        {       capDriverConnect(hCam, 0    );   // 0 = Default driver
                capPreviewScale (hCam, false);
                capPreviewRate  (hCam, 66   );
                capPreview      (hCam, false);
        }
}
```

Changes in the bitmaps affect the availability of the related abilities instantly, both native and shared, in the user interface. This is very useful because shared abilities can become available/unavailable dynamically and in an uncoordinated way, e.g., when systems enter/leave the SoS respectively. Native abilities can also be set available/unavailable anytime as modeled in the SoS use case diagram of Figure 35, e.g., “Change Set of Calculations Available”. Monitoring the bitmaps enables the user interface to be always up to date.

At this point we have a traditional SPL able to deploy several instances of calculators with a different set of native features. Every feature was assigned to a particular component and saved in the SPL core assets. Next, we exemplify the component of the feature + (Figure 33):

```
int addFeature(double p1, double p2, double* res)
{       *res  = p1 + p2;
         return 0;
}
```

Different from many SPL implementations that add components at design time to enable optional features, all components are assembled in the code and their availabilities are managed by the `localOp` bitmap as described above. This enables optional features to be enabled/disabled dynamically at runtime.

The evolution of the SPL to deploy SoS members was performed with the guidance of the SoS PCB-Statechart of Figure 61 and the SPLE extended process of Figure 62. The PCB-Statechart was useful to define which processes should run concurrently, their assignments, and how they should interact.

Particularly in this case study, each component relates to a unique ability. Thus, they were easily extended to include interaction capabilities. The following example shows the `addFeature()` component shown previously extended to meet the SoS requirements. Thus, it is now rather an ability than a component and will compose the FoS core assets.

```
int addAbility(double p1, double p2, double* res)
{
		if(isAvailable_Local (_ADD_)) return addFeature(p1, p2, res);
	else	if(isAvailable_Shared(_ADD_))
	{
		return soap_call_ns__add(&soap, Endp[_ADD_],"", p1, p2, res);
	}
	else	return = -1; // Failed
}
```

The support for services was implemented by adapting the gSOAP toolkit to our needs (gSOAP, 2013). In the above component, if addition is a native ability, it will be executed as usual by the `addFeature()` component. Otherwise, if it is a shared ability discovered during compositions, the execution will be forwarded to the SoS member that hosts the ability (located at the *Endpoint* informed during the composition process, Figure 41). Concurrently, each SoS member will be listening for requests as modeled in the PCB-Statechart of Figure 61 (e.g., *Calculation* state) and described in the SoS requirements (Section 4.4). The thread loop looks like the following:

```
m =	soap_bind(&soap, NULL,_PORT_, 100);	// Bind to the service port
if(	soap_valid_socket(m) )
{	soap.accept_timeout= 1;
	do
	{	s =	soap_accept	(&soap );	// Wait for service requests
		if(	soap_valid_socket(s))
		{	soap_serve	(&soap );	// Provide service
			soap_end	(&soap );
		}
	}
	while(!thread->exit);
}
soap_done(&soap);
```

In the above loop, whenever a shared ability is requested by other SoS member, `soap_serve()` will forward the call internally to the appropriate function. For example, if an addition is requested and it is a native ability, then the following function will be called:

```
int ns__add(struct soap *soap, double a, double b, double *result)
{
	*result = addFeature(p1, p2, res);
	 return	SOAP_OK;
}
```

It is important to notice that the component `addFeature()` of the SPL core assets was used to perform an addition exactly as in `addAbility()` (an ability of the FoS core assets). However, the circumstances are different. In `addAbility()` it performs additions as a native ability. On the other hand, in the thread that listens for requests from other systems (a

component of the FoS core assets), it performs additions as a shared ability in behalf of the requesters. This fully complies with the extended SPLE process of Figure 62 as follows.

Components of the SPL core assets (e.g., `addFeature( )`) do not establish dependency relationships neither with abilities (e.g., `addAbility( )`) nor with components (e.g., listeners) of the FoS core assets, thus the SPL can still instantiate products to operate as originally designed (independent systems). However, the vice-versa is desirable, i.e., abilities and components of the FoS core assets strongly reuse components of the SPL core assets to increase productivity and reliability. Abilities and components are different concepts and are maintained separately in the FoS core assets. Indeed, components make effective the sharing of abilities in the SoS as discussed previously.

Finally, part of the SoS contract is shown in the following. It was partially generated by the gSOAP tollkit and describes the addition ability as a service.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="calc"
    xmlns:tns ="http://websrv.cs.fsu.edu/~engelen/calc.wsdl"
    xmlns:xsd ="http://www.w3.org/2001/XMLSchema"
    xmlns:wsa5="http://www.w3.org/2005/08/addressing"
    xmlns:wsdd="http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01"

    <message  name="addRequest">
        <part name="a" type="xsd:double"/><!-- ns__add::a -->
        <part name="b" type="xsd:double"/><!-- ns__add::b -->
    </message>

    <message  name="addResponse">
        <part name="result" type="xsd:double"/><!-- ns__add::result -->
    </message>

    <message  name="calcHeader">
        <part name="MessageID"   element="wsa5:MessageID"/>
        <part name="RelatesTo"   element="wsa5:RelatesTo"/>
        <part name="From"        element="wsa5:From"/>
        <part name="ReplyTo"     element="wsa5:ReplyTo"/>
        <part name="FaultTo"     element="wsa5:FaultTo"/>
        <part name="To"          element="wsa5:To"/>
        <part name="Action"      element="wsa5:Action"/>
        <part name="AppSequence" element="wsdd:AppSequence"/>
    </message>

    <portType name="calcPortType">
        <operation name="add">
            <documentation>Sums two values</documentation>
            <input  message="tns:addRequest"/>
            <output message="tns:addResponse"/>
        </operation>
    </portType>

    <binding name="calc" type="tns:calcPortType">
        <SOAP:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="add">
            <SOAP:operation style="rpc"/>
            <input>
                <SOAP:body use="encoded" namespace="urn:calc"
encodingStyle="http://www.w3.org/2003/05/soap-encoding"/>
```

```
            </input>
            <output>
                <SOAP:body use="encoded" namespace="urn:calc"
encodingStyle="http://www.w3.org/2003/05/soap-encoding"/>
            </output>
        </operation>
    </binding>
</definitions>
```

In Figure 65 we show the calculators of Figure 63 interacting in the SoS environment.

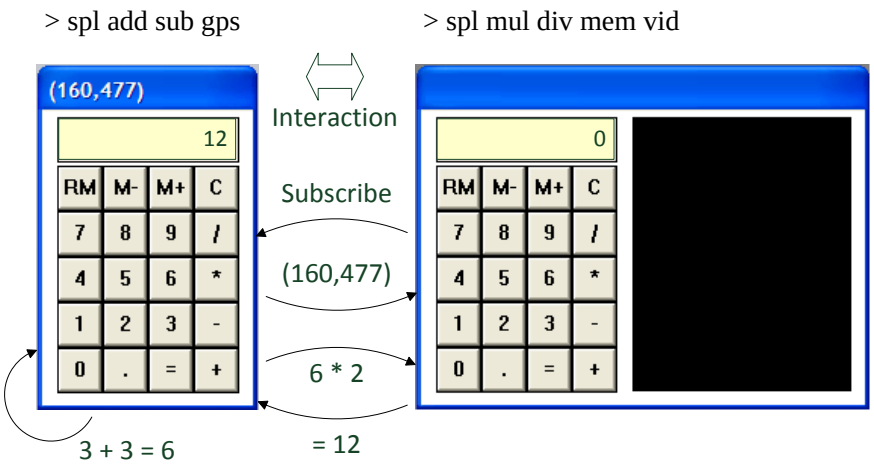


Figure 65. Two different SPL instances interacting in the SoS environment

6.4. *Final considerations*

Several approaches have been proposed for SPLE and SoSE separately. But, to the best of our knowledge, making them to work complementarily has not been explored. For this reason, we intentionally did not include a separate section to discuss related work.

Although we have frequently exemplified the use of services to support systems' interactions, our SoSE and SPLE approaches are not limited to them. Actually, the best interaction mechanism may be different for distinct scenarios. SoS engineers are responsible for analyzing the options available and decide which are the most appropriate for their SoS.

Services can support interactions in a large number of solutions. However, this term is still strongly tied to SOA (Erl, 2008) as SoS is tied to complex systems (Maier, 1998). Thus, the same effort applies to give the term a less restrictive concept. Actually, elements like service repositories can be very restrictive when we are attempting to make systems interact naturally in their regular environments.

Although SoS is a promising paradigm for solving multidisciplinary problems and SPL has a great potential to support it, we argued that there is still a gap between SPLE and SoSE to be overcome until they can deliver meaningful results jointly. The SPLE extensions that we proposed (see Figure 62) suggests that concerns about SoS characteristics should be

considered throughout the SPLE process to ease later evolutions of SPL products to SoS members. Moreover, SPL and SoS artifacts must relate closely to each other but evolve separately to preserve the productivity and profitability of the SPL products as independent systems.

The case study performed in Section 6.3 highlighted the importance of detailed SoS requirements to the success of SoS initiatives. However, as demonstrated, the focus should not be the SoS members but their interactions instead. Indeed, we have not mentioned a particular system throughout the requirements activities (*Skecth*, *Detail* and *Organize*) of the SoSE process (Figure 23). Only when the SoS requirements were fully modeled and described a candidate system was proposed and developed. As expected, it was able to compose the SoS harmoniously.

Dividing the SoS goal in scenes (Figure 36) was very important to build it correctly. For example, '*Enter SoS*' was the first scene addressed. The focus was the components of the FoS core assets like the listener responsible for processing discovery messages. Next, the '*Leave SoS*' scene took place. Then, the SoS dynamism could be fully exercised before a single ability has been deployed.

Although the use cases modeled and described in Section 4.4 were useful to identify interaction points and understand related systems' behaviors, it was the SoS PCB-Statechart of Figure 61 that really helped on the design of the appropriate system's structure to better handle interactions. Indeed, listeners and service-based components of the FoS core assets were designed to fit accurately the structure of the PCB-Statechart.

As discussed above, two extensions we proposed to traditional RE approaches were important to the sample SoS to succeed. This fact highlights that traditional RE techniques are necessary, but may not be sufficient to fully support SoS projects (Meyers et al., 2006).

Finally, the case study performed highlights that virtually any networked system can become an SoS member. This enables SoS engineers to design SoS systematically rather than opportunistically and expand the number of solutions that can be solved by this emerging paradigm. However, different of our sample virtual SoS, it would be necessary to consider the reuse of legacy systems in most of the cases. Next, we propose a reengineering approach to help on this matter.

Chapter 7

A reengineering approach extension to SoS

7.1. Initial considerations

Nowadays, there are a countless number of software systems running worldwide to assist people in their daily activities. Many of them might have abilities that would be useful to other systems to improve some particular functionality. However, these abilities often are not made available by the host systems to the potential clients.

Some abilities may be complex enough or depend on particular environment properties to work adequately. An example is the resources of the IOOS (2013) of Figure 8. In this case, these abilities can hardly migrate from one system to another, for example, like COTS in component-based systems (Heineman and Councill, 2001). Thus, when they are intended to be shared among systems, appropriate interaction mechanisms and standard interfaces must be commonly defined, e.g., services and contracts in SOA (Erl et al, 2008).

Even though services represent an improvement toward a better integration of systems and their abilities (Erl, 2009), engineers still have to overcome the challenge of evolving their legacy systems to conform to the service-oriented vision. Again, an important distinction should be made between service-orientation and SOA as discussed in Section 2.5.

Evolving legacy systems to service-oriented can speed up SoSE processes since useful abilities that are currently restricted to their systems can be shared and reused. Of course, the feasibility of this process depends directly on how easy it is to evolve these systems to meet SoS requirements. The challenge is increased when the interoperability relationships between SoS members go beyond information exchange and lay on open interaction interfaces.

When legacy systems are required to interact more actively and to behave distinctly in different SoS, the successive reengineering of these systems to meet those SoS requirements may be complex and also affect negatively their maintainability.

The reengineering approach described in this section results from our effort toward the design of service-oriented SoS from independent systems of different domains. In the current

phase, we aim to build families of system (FoS) from a variety of legacy systems, including those that were not originally designed to share abilities.

This chapter is organized as follows. In Section 7.2, we discuss related work. In Section 7.3, we propose a reengineering approach extension to SoS so that legacy systems can become SoS members. In Section7.4, we evolve the case study of Sections 4.4, 5.5 and 6.3 by deploying members to the virtual SoS from legacy systems. Finally, in Section 7.5, we present our final considerations.

7.2. Related work

Ramos and Penteado (2008) proposed a reengineering approach to SPL that supports this part of the thesis. It mixes concepts and activities of incremental reengineering and SPLE. Useful functions of a legacy system are identified, modeled as features, mapped, mined, and reengineered as components to populate SPL core assets. Figure 66 illustrates this process.

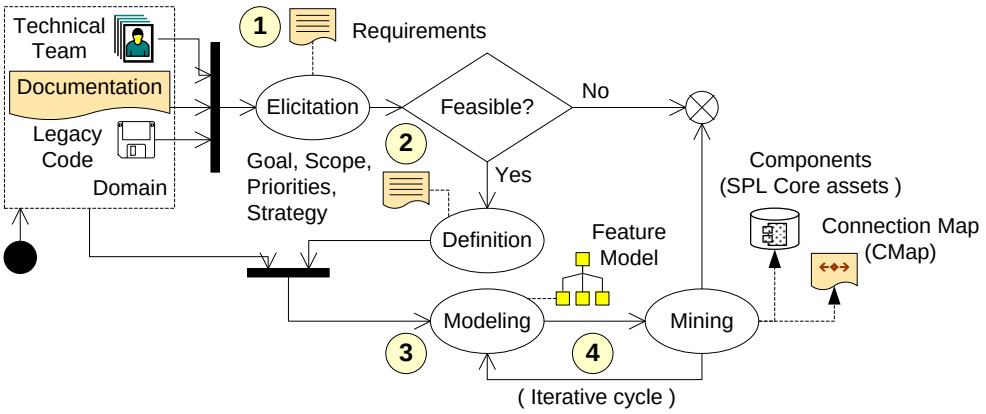


Figure 66. Reengineering approach to SPL (Ramos and Penteado, 2008)

Four activities are performed: (**1**) A kickoff meeting gathers people involved with the legacy system to elicit the requirements of immediate and future improvements; **(2)** A feasibility analysis is performed. If a decision is taken favorable to the continuity of the process, the requirements will support the outlining of the strategy and the definition of the scope, goals, and priorities of the reengineering; **(3)** Technical people are gathered to model the features embedded in the legacy system. Feature models are used to support this activity. Depending on the method used to decompose the legacy system, features may be of different nature such as strategic functions (functional) or hardware components (structural); and **(4)** An incremental reengineering activity is performed. The legacy code is inspected and the functions associated with each feature are mined, mapped in a Connection Map (CMap), moved to components, and stored in the SPL core assets iteratively. This must be done in such

a way that the maintainability of the legacy system is either not affected or reduced as little as possible.

The CMap artifact maps each feature to a set of related functions in the legacy code. The prototype of these functions outlines the interface of the component that fully implements the feature as well as the interface of the gateway that will reintegrate the component in the legacy code. The gateway enables engineers to develop the components using more modern programming languages and paradigms.

The mined components are reintegrated logically in the legacy code through gateways and will improve the system with variability mechanisms. Surrogate codes are developed to forward original function calls to the gateway and thereafter to the components. Techniques like inheritance and parameterization are applied to the components to deploy optional and alternative features and then a family of related products. Figure 67 illustrates this process.

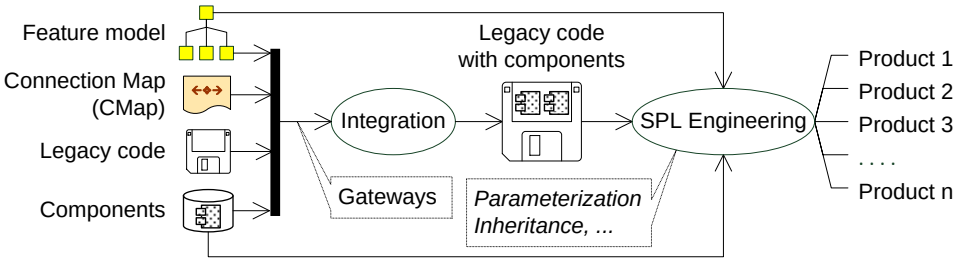


Figure 67. Instantiation of SPL products (Ramos and Penteado, 2008)

7.3. The proposed reengineering approach extension

The proposal to extend the reengineering approach to SPL of Figure 66 intends to improve the reengineered systems with service capabilities thus they can be grouped into FoS and reused to compose SoS, as required by the SoSE process of Figure 23.

One of the challenges of reusing FoS members to compose service-oriented SoS is that different SoS may require distinct services and service interfaces over time. Thus, it may be necessary to create services on demand according to the current SoS requirements. Moreover, services must not be coupled with the legacy code to preserve the maintainability of the code and to increase the adaptability and reusability of the services.

Another challenge on achieving our objective is that the reengineering must not change the regular operation of the legacy systems or create dependency relationships with other systems (systems' autonomy, Table 1). Figure 68 illustrates the approach extension we propose to handle these challenges.

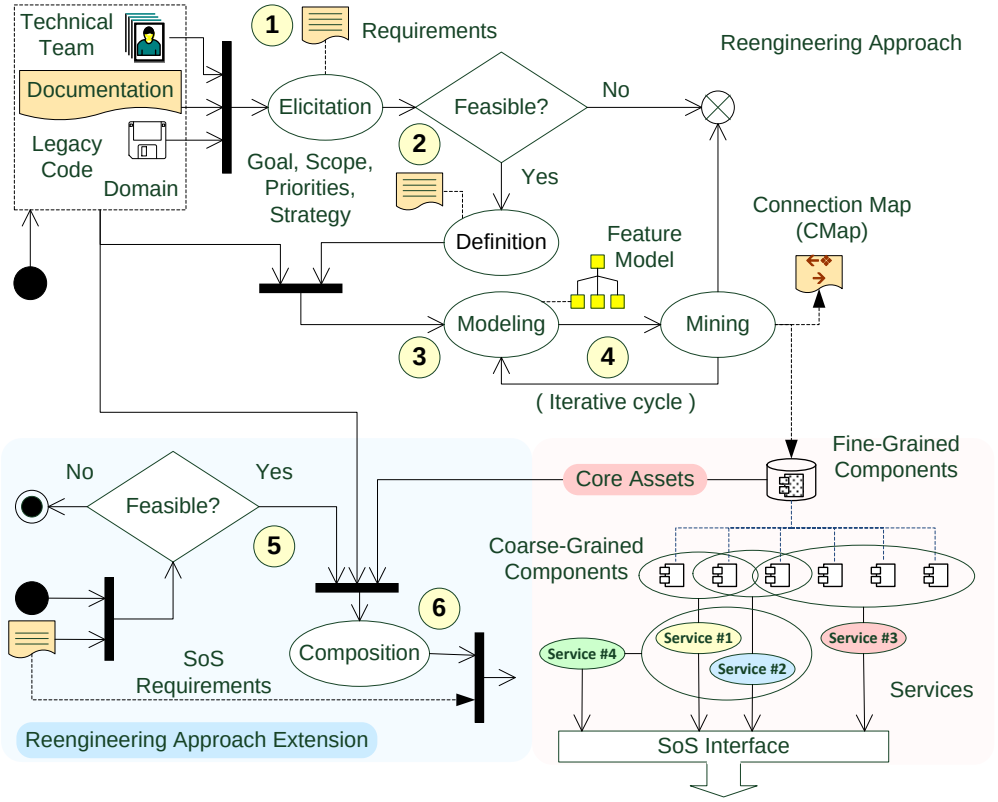


Figure 68. Reengineering approach extension

Firstly, the reengineering approach to SPL is performed to mine incrementally useful functions present in the legacy code. The components that result from this activity often will be fine-grained, thus they can be more generic and reusable. They are intended mainly to be building blocks for coarse-grained components and services, which can be maintained or developed on demand to meet different SoS requirements. Every new mined component is stored in the core assets repository to be reused opportunely.

Even though services are important to the context of this thesis, the most critical elements are still the fine-grained components. Without a comprehensive set of representative components, systems engineers will probably face difficulties to develop useful services to SoS. Furthermore, because of the gateways, components are loosely coupled with the legacy code, thus services will be either, as required.

Two new activities were added to the original reengineering approach to SPL (at the bottom of Figure 68):

1. A feasibility analysis is performed based on the SoS requirements which describes, for example, collaboration rules and interactions mechanisms. If the required SoS features do not exist neither in the core assets repository nor in the legacy code the system is potentially not a candidate system. Otherwise, the features are matched against the ones available in the

core assets repository. If they fully match or partially match and the necessary maintenance is feasible, then the constituent systems can be deployed. If they are available only in the legacy code, then they must be mined first, and the feasibility will be decided based on the results of the reengineering process.

2. The fine-grained and coarse-grained components and the services selected in the previous activity must be reused as is wherever possible. Otherwise, they must be developed or maintained to meet the current SoS requirements. By development, we mean joining components and services to create more complex functionalities. If this activity succeeds, the SoS interface directs the system to behave as expected in the SoS.

7.4. Case study – Composing the sample virtual SoS with legacy systems

At this point, our sample SoS has already a couple of members, each representing an instance of the calculator SPL described in Section 6.3, which meet the SoS requirements described in Sections 4.4 and 5.5. This case study was performed to demonstrate that legacy systems from different domains can be reengineered to become a member of the sample SoS as well (Ramos et al., 2012). This process aims to play the role of supplier in our SoSE process (Figure 23), similarly to the extended SPLE process proposed in Section 6.2 (Figure 62). Particularly in this case study, a Point Of Sale (POS) legacy application was chosen because the POS development platform supports ANSI C, thus most of the gSOAP (2013) code developed in the previous case study (Section 6.3) could be reused, e.g., listeners to requests and discovery messages.

POS terminals are portable and versatile devices that assemble a reduced set of standard peripherals, managed by an embedded operating system. They are mostly used to accomplish Electronic Funds Transfer (EFT) with payment cards. Figure 69 illustrates an example of POS terminal and its usual hardware peripherals (Verifone®, 2013).

The POS legacy application we used belongs to the company Verifone® (2013) and its details are protected by non-disclosure agreements[1], thus we will refer to it just as POS-App. It is fully independent and was designed to run standalone. Moreover, it has monolithic and procedural code written in ANSI C. Conditional blocks of code managed by preprocessor statements (#if, #endif) are the only variability mechanisms available.

[1] The author is currently employed as software engineer at the company Verifone Inc.® and had authorized access to the application source code exclusively for this case study. However, code details could not be revealed due to copyrights.

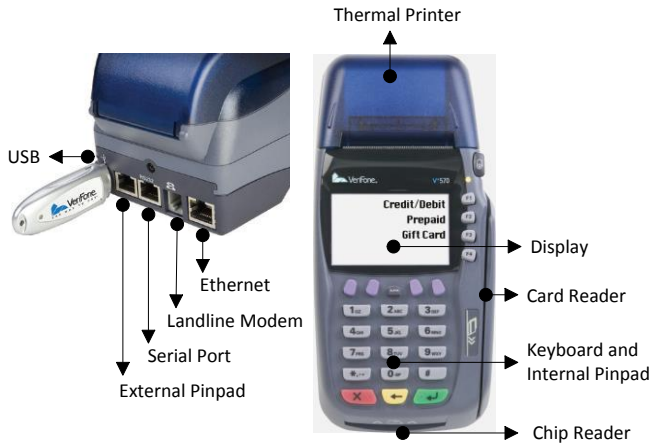


Figure 69. POS terminal and its usual HW peripherals (Verifone®, 2013)

The SoS requirements (see Sections 4.4 and 5.5) supplied the information necessary to start the reengineering process (Figure 68 – 1). To support the feasibility analysis (Figure 68 – 2), we performed a combined structural and functional decomposition of the POS terminal to define the reengineering strategy and scope. The resulting feature model is shown in Figure 70. Considering the scene-based RE approach proposed in Section 4.3, this activity fits in the Low-Level Requirements – Member Systems phase of Figure 28.

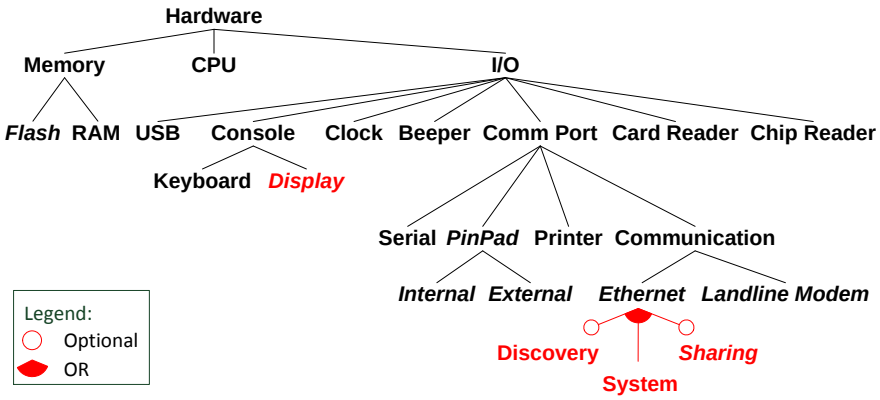


Figure 70. POS terminal feature model

Additionally, the POS-App code was analyzed and the architecture model of Figure 71 was drawn.

In the iterative cycle (Figure 68 – 3,4) the structural feature *Display* and the functional features *Discovery*, *System* and *Sharing*, were the target features to be reengineered. The second should increase the communication capability of the legacy code, necessary to support interactions. The mandatory feature *System* represents the current communication capability of the system and must support its regular operation as an independent system. The optional

feature *Discovery* represents a new communication capability necessary to manage dynamic compositions. The optional feature *Sharing* represents another new communication capability necessary to support the sharing of abilities when belonging to the SoS.

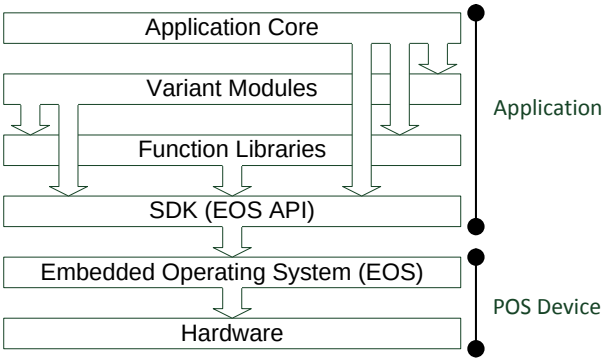


Figure 71. POS-App architectural model

Due to security reasons, it was not allowed to the POS-App to share abilities with the current SoS members. Exceptionally, it can subscribe to receive information published in the SoS. Particularly in this case study, it receives information about location and displays it on the POS display.

The structural features did not require any mining activity to be implemented because of the architecture of the legacy code. Two function libraries were used by the legacy code to implement TCP/IP communication through Ethernet. The first enables the sharing of the Ethernet hardware among threads by taking its physical handle and managing logical handles. The second supports concurrent communication through multiple sockets (TCP/UDP). This way, listeners were implemented using concurrent threads and the gSOAP (2013) toolkit, similarly to those described in Section 6.3. Thus, the POS terminal was able to compose the SoS quickly and subscribe to receive updated location coordinates periodically. Belonging or not to the SoS, the POS-App was always able to operate independently.

The *Display* feature was mined because of its dependency relationship with the legacy code. Indeed, there were numerous calls to display functions throughout the code. A problem to solve was that the legacy code was not designed to share the display, thus concurrent threads would not be able to use it. The reengineering of the legacy code should provide a way to share the display among threads. Next, we summarize how it was done by applying the activities proposed by Ramos and Penteado (2008), illustrated in Figure 66 and Figure 67.

Firstly, all functions directly related to the display that affect the reengineering goal were documented in a Connection Map (CMap). The result is shown in Table 7.

Table 7. Display feature Connection Map (CMap)

Feature:		**IO → Console → Display**	
Function	**Parameters**	**Return**	**Comments**
open	Console ID	Console HND: int	Fail (< 0) if the console is already opened
close	Console HND	Status: int	Free the console handle

[Constraints]
. Display can only be shared among a main process and its child threads if explicitly required
[Requirements]
. The console must be shared among threads

Then, the display component was created and the codes of the mapped functions were moved to the corresponding methods into the component. In the legacy code, all references to those functions moved to the component received the suffix _G_ (of Gateway), i.e., they become *_G_open* and *_G_close* function calls. Next, the gateway was similarly created with the guidance of the CMap. The only functions implemented were *_G_open* and *_G_close*, which simply forward calls to the corresponding methods in the display component, i.e., *open* and *close* respectively. Finally, the gateway and the component were compiled with the legacy code to create the reengineered version of the POS-App. The resulting POS-App architecture is shown in Figure 72 and the code changes are exemplified next.

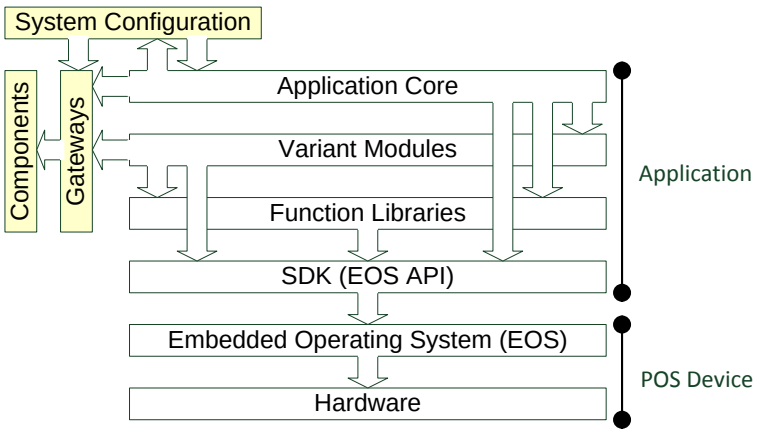


Figure 72. POS-App new architectural model

```
// Legacy code
// Function call
if( _G_open("/dev/console", 0) <  0 )   // Renamed
{ /* Err */ }

/* Function implementation */
int open(char* devname, int mode)       // Removed
{ ... }
// Gateway
int _G_open(char* devname, int mode)
{
      return Display::open(devname, mode); // Forward the call to the component
}
```

```
// Component
public class Console
// Console interface
{ ... public: virtual int open(char* devname, int mode) = 0;
}

public class Display: public Console
{ ...
       public: int open(char* devname, int mode);
}
Display:: open(char* devname, int mode)
{
       // Check system configuration
       // If enabled SoS { Start threads and share the display }
 ...
}
```

As a result, the component could improve the original methods without affecting the legacy code structure. Particularly in this case study, the component enabled the display to be shared among all child threads whenever it is opened by the POS-App. At this moment, the listeners and the thread responsible to display location coordinates are started. Whenever the display is closed, only the listener to discovery processes are kept alive. However, a new system configuration can be set to disable the system to compose SoS. In this case, no threads are started and the system behaves exactly as the original.

It is important to notice that the only change observable in the legacy code is the _G_ suffix of some functions. Thus, the maintainability was fully preserved.

7.5. Final considerations

In this section, we considered legacy systems to compose SoS because of the countless number of these systems that are currently running worldwide without sharing abilities. Also, we described how basic but useful features can be mined from the legacy code, moved to components, and reused repetitively to deliver distinct functionality. If the proposed reengineering approach is applied iteratively, the core assets repository can be populated increasingly to support SPL and SoS development.

The case study we performed revealed that the reengineering approach extension we proposed (Figure 68) could not always be followed as is. Two examples were given. Listeners responsible to handle compositions and the sharing of abilities could be developed completely uncoupled from the legacy code because of the hardware and software architecture, and the application constraint that does not allow the sharing of abilities for security reasons. Moreover, since the system does not share abilities, the service interface the way it was drawn in Figure 68 was not necessary to compose the sample virtual SoS. However, we think this

was a particular case, but yet meaningful, in which a system becomes an SoS member to get benefits, not to benefit.

It sounds clear that the proposed reengineering approach (Figure 68) can be a natural first step for the proposed SPLE extensions (Figure 62). Indeed, not all legacy systems belong to SPL. However, if they are reengineered to SPL first, then both approaches can be applied complementarily.

Chapter 8

Conclusions

8.1. Summary

With the guidance of a sample virtual SoS that comprises all the distinguishing SoS characteristics, this thesis explored the development of an SoS from different SE perspectives. For each of them, we discussed constraints of conventional SE approaches and proposed extensions to them to support SoS development. An SoSE development process was drawn to fit all the proposed SE approach extensions. A scene-based RE approach was proposed to describe an SoS progressively as an arrangement of meaningful behaviors named 'scenes'. Symbolic notations resulting from an analogy with PCB was proposed to extend statecharts and to visually improve the modeling of interactions among systems. The traditional SPLE process (Ziadi, Jézéquel, and Fondement, 2003) was extended with new activities related to SoS development so that an SPL can become a natural source of SoS members. Finally, a reengineering approach extension was proposed so that legacy systems can share useful abilities, work cooperatively, and compose SoS.

8.2. Contributions

A short time after deciding to build a sample virtual SoS we realized that we could not point to a single direction into SoSE that did not touch a problem without an accepted solution. Indeed, even definitions of SoS seemed to not converge. Finally, the SoS definition and scope that we considered seemed to have never been proposed in the literature. Thus, to deploy the sample virtual SoS it was necessary to use conventional SE approaches but with some extensions. More than that, it was necessary to propose an SoSE process able to fit together those approaches and extensions complementarily. This SoSE process and the SE approaches extensions are the main contribution of this thesis toward the development of SoS.

The proposed scene-based RE approach (Section 4.3) can ease the description and the understanding of the SoS dynamism. Statechart extensions (Section 5.2) can improve visually the modeling of interactions among systems. The extended SPLE process (Section 6.2) can make SPL a natural source of SoS members. Finally, the reengineering approach to SPL and

SoS (Section 7.3) can improve legacy systems with new features and make possible to those systems to share abilities in SoS environments.

Unlike related work involving SoS, this thesis explored virtual SoS and its properties. We discussed and exemplified the development of an environment in which SoS members can freely collaborate to improve their own purposes and achieve supra-purposes, i.e., SoS goals. We demonstrated that an SoS can be an abstract concept and just a reference to the systems collaborating in such environment. Indeed, neither a coordinator nor glue code was necessary to deploy the sample virtual SoS. Moreover, we exemplified that an SoS may have multiple classifications considering different sets of systems.

To the best of our knowledge, none of the above topics have been exemplified in the literature. Moreover, we did it by composing even small and simple conventional systems. Thus, we think that this thesis contributes to the research of SoS by pointing to alternative directions in which ordinary systems can collaborate to achieve goals currently achieved by single systems.

8.3. Constraints

In this thesis we adopted a comprehensive SoS definition, i.e., a composition of virtually any system able to contribute to the SoS goal. To evolve this concept, we focused on the distinguishing SoS characteristics of Table 1 (Boardman and Sauser, 2006) rather than on the characteristics of its members, e.g., complex systems. This way, the solutions we proposed possibly may not apply to the examples of SoS commonly present in the literature. For example, information-based SoS like the GEOSS (Figure 1) and the IOOS (Figure 8) contain transversal systems developed to gather and format information from a variety of sources and deliver high quality data and information. This approach creates dependency relationships among transversal (consumers) and vertical (providers) systems decreasing their autonomy. Indeed, transversal systems may not even exist out of the SoS. This example can explain why virtual SoS like the one exemplified in our case study is rarely exemplified in the literature.

Although the sample SoS that base our case study comply with all the distinguishing SoS characteristics, it may not be comprehensive enough to reveal all the variant scenarios that can emerge when real systems are composed. However, it was necessary to start with a possibly ideal scenario to propose and validate new approaches or extensions toward the development of SoS before going further with more comprehensive solutions.

Finally, strengths and constraints of the proposed approaches can only be stated when they have been widely used to support the development of different SoS solutions. This is part of future work, described next.

8.4. Remarks and Future work

If PCB-Statecharts, proposed in Section 5.2, were supported by modeling tools, then systems' interactions, either simple or complex, could be tested and simulated before any system is designed or definitively chosen to compose an SoS. Work on such tools and their applications to support SoS projects is part of future work. However, we must provide first some formalism to PCB-statecharts. Additionally, we have to investigate how easily the proposed symbolic notations discussed in Section 5.2 can be understood and applied by students and engineers to describe interactions. Finally, we must stimulate researchers to use PCB-statecharts in different software engineering areas and projects to analyze extra benefits and constraints that we have not discussed in this thesis.

The case study of Section 7.4 reinforces our statement that virtually any system can become an SoS member. Although it is desirable, not all systems will share abilities naturally for example, unless agreed security policies are applied. However, this will certainly affect the whole SoS requirements, e.g., cyphered messages, secured communication channels etc. This scenario was not handled in this thesis. However, all the presented proposals can still be applied due to their generality. Researching variants of the presented scenarios and their impacts in the SoS context is part of future work as well.

Addendum

The sample SoS

This addendum shows the first sketch of the sample SoS detailed in the case studies of this thesis (Sections 4.4, 5.5, 6.3, and 7.4). The text below was written in the very beginning to support discussions about SoS and its distinguishing characteristics (Table 1) related to conventional systems. Three roles are supposed to be played by the member systems: calculator, publisher, and processor. Next, we describe the first requirements of the possible players.

The calculator

- It must always be able to run autonomously;
- It must be able to perform calculations;
- It is network-enabled;
- It must be able to compose dynamically the SoS;
- It must be able to share calculations, i.e., perform calculations in behalf of other systems;
- The implementation of each calculation must be independent and arbitrary;
- Calculations are optional features;
- At least one calculation must be active at run-time;
- The set of calculations available is {+,–,∗,/,M+,M–,MR}.

The publisher

- It must always be able to run autonomously;
- It must be able to collect information;
- It is network-enabled;
- It must be able to compose dynamically the SoS;
- It must be able to handle subscriptions;
- It must be able to publish information to subscribers;
- It must be able to compose SoS;

The processor

- It must always be able to run autonomously;

- It must be able to process information;
- It must be able to deliver useful functionality;
- It is network-enabled;
- It must be able to compose dynamically the SoS;
- It must be able to request for information;
- It must be able to compose SoS;

Member systems

- A system can optionally play multiple roles simultaneously;
- Each role can occasionally, independently, and dynamically be enabled or disabled;

SoS characteristics

- No dependency relationships must be created among systems (Autonomy);
- Systems will compose the SoS dynamically to share abilities and get benefits (Belonging);
- Emergent behaviors are expected to happen (Emergency);
- A role can be played by different systems to increase redundancy (Diversity);
- System can interact directly with each other (Connectivity);
- New missions can be assigned to the SoS (Evolutionary);
- System can enter/leave the SoS in an uncoordinated way (Dynamism).

The resulting SoS

The resulting SoS with two member systems is shown in Figure 73 and Figure 74. System#1 plays the roles of Calculator and Processor. System#2 plays the roles of Calculator and Publisher. Although each system implements a subset of calculations (because of the command lines that started them), both can do all calculations available because of the SoS. Moreover, System#1 process continuously the location information provided by System#2 to decide when the camera should start capturing video (the mission) (Figure 73). Both systems compose the SoS dynamically, run autonomously, and perform the mission collectively. The SoS conforms to all the above SoS characteristics as discussed in the previous chapters.

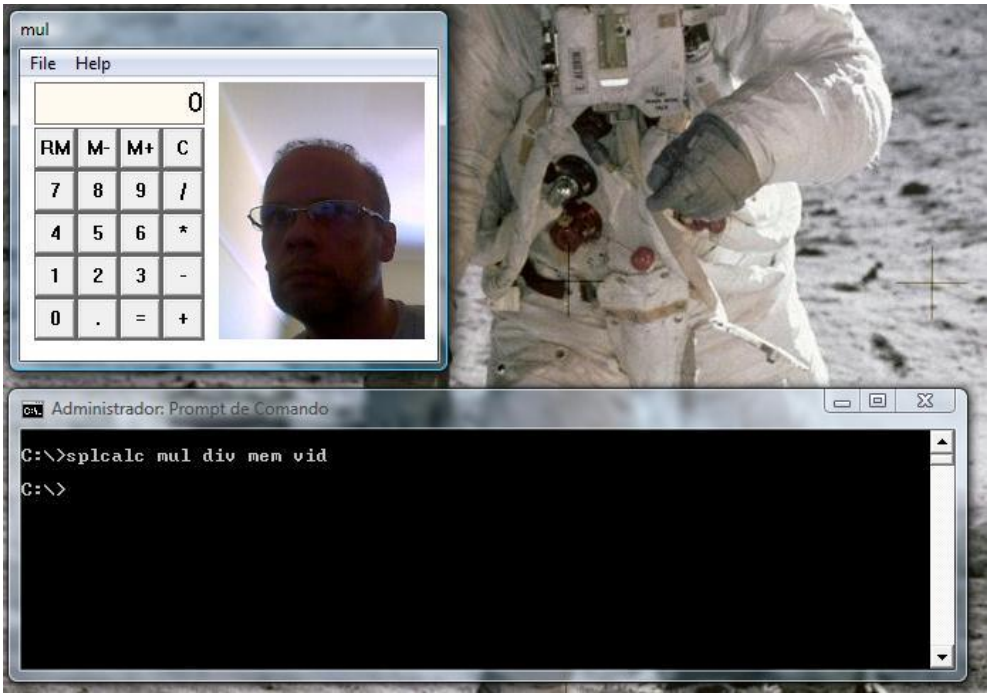


Figure 73. System#1 – Calculator + Processor

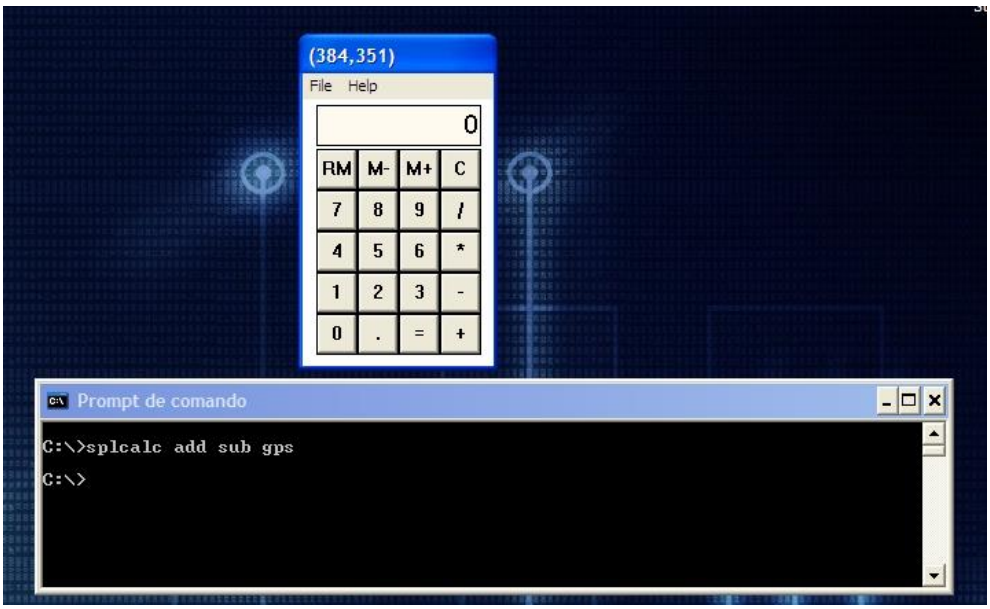


Figure 74. System#2 – Calculator + Publisher

References

ATKINSON C., BAYER J., BUNSE C., KAMSTIES E., LAITENBERGER O., LAQUA R., MUTHIG D., PAECH B., WUST J., ZETTEL, J., 2001. "*Component-Based Product Line Engineering with UML*". Addison-Wesley.

ATZORI L., IERA A., MORABITO G., 2010. "*The Internet of Things: A survey*". Journal Computer Networks: The International Journal of Computer and Telecommunications Networking, 54(15): pp.2787-2805.

AZANI C., 2009. "*An Open Systems Approach to System of Systems Engineering*". In System of Systems Engineering: Innovations for the 21st Century, Mo Jamshidi, John Willey & Sons Inc.

AZANI, C.H., KHORRAMSHAHGOL, R., 2005. "*The open system strategy: an integrative business and engineering approach for building advanced complex systems*". Proceedings of the 9th World Multiconference on Systemics, Cybernetics and Informatics (WMSCI, 2005), Orlando, FL

BAEK S., HAN J., CHUNG K., 2013. "*Dynamic Reconfiguration Based on Goal-Scenario by Adaptation Strategy*". Wireless Personal Communications, pp. 309-318.

BERGE C., 1973. "*Graphs and Hypergaphs*". North-Holland Publ. Co.

BOARDMAN J., SAUSER B., 2006. "*System of Systems - the meaning of of*". IEEE/SMC International Conference on System of Systems Engineering, pp.118-123.

BOOCH G., RUMBAUGH J., JACOBSON I., 1999. "*The Unified Modeling Language Reference Manual"*, Addison-Wesley, USA.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I., 1999. "*The Unified Modeling Language Reference Manual*", Addison-Wesley, USA.

BROWNSWORD L., FISHER D., MORRIS E. J., SMITH J., KIRWAN P. 2006. "*System-of-Systems Navigator: An Approach for Managing System-of-Systems Interoperability*". SEI Technical Note CMU/SEI-2006-TN-019. Available at: http://resources.sei.cmu.edu/asset_files/TechnicalNote/2006_004_001_14699.pdf

BUSCHMANN F., MEUNIER R., ROHNERT H., SOMMERLAD P., STAL M., 1996 "*Pattern-Oriented Software Architecture Volume 1: A System of Patterns*". Wiley

CARLOCK P.G., FENTON R.E., 2001. "*System of Systems (SoS) enterprise systems engineering for information-intensive organizations*". Systems Engineering, 4(4): pp.242-261.

CETINA C., FONS J., PELECHANO V., 2008. "*Applying Software Product Lines to Build Autonomic Pervasive Systems*", 12th International Software Product Line Conference (SPLC '08), pp.117-126.

CLARK J.O., 2009. "*System of Systems Engineering and Family of Systems Engineering from a standards, V-Model, and Dual-V Model perspectiv*e". 3rd Annual IEEE Systems Conference, pp.381-387.

CLEMENTS P., NORTHROP L., 2001. "*Software Product Lines: Practices and Patterns*", Addison-Wesley.

DRAGANFLY, 2014. "*Innovative UAV Aircraft & Aerial Video Systems*". Available at: http://www.draganfly.com

ERL T., 2008. "*Service-Oriented Architecture. Concepts, Technology, and Design*". Practice Hall.

ERL T., 2009. "*SOA Design Patterns*". Practice Hall, USA.

ERL T., KARMARKAR A., WALMSLEY P., HAAS H., YALCINALP U., LIU C. K., ORCHARD D. U., TOST A., PASLEY J., 2008. "*Web Service Contract Design and Versioning for SOA*". Prentice Hall.

GEOSS, 2013. "*The Global Earth Observation System of Systems*". Available at: http://www.earthobservations.org/geoss.shtml

GLINZ M, 2002. "*Statecharts For Requirements Specification – As Simple As Possible, As Rich As Needed*". In: Proceedings of the ICSE 2002 International Workshop on Scenarios and State Machines: Models, Algorithms and Tools. Available at: https://files.ifi.uzh.ch/rerg/arvo/ftp/papers/SCESM2002.pdf

GOMAA H., 2004. "*Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*". Addison-Wesley, USA.

gSOAP, 2013. "*The gSOAP Toolkit for SOAP Web Services and XML-Based Applications"*. Available at: http://www.cs.fsu.edu/~engelen/soap.html

HAREL D., 1988. "*On visual formalism*". Magazine Communications of the ACM, 31(5): pp.514-530, ACM New York, USA.

HAREL, D., 1987. "*Statecharts: A visual formalism for complex systems*". Journal Science of Computer Programming, 8(3): pp.231-274.

HEINEMAN, G. T.; COUNCILL, W. T., 2001 "*Component Based Software Engineering: Putting the Pieces Together*". Addison-Wesley, USA.

HOPCROFT J.E., ULLMAN J.D., 1979. "*Introduction to Automata Theory*". Languages and Computation. Addison-Wesley.

HUYNH, T.V., OSMUNDSON, J.S., 2006. "*A systems engineering methodology for analyzing systems of systems using the systems modeling language (SysML)*". In: Proceedings of the 2nd Annual System of Systems Engineering Conference, Ft. Belvoir, VA, sponsored by the National Defense Industrial Association (NDIA) and OUSD AT&L. Available at: http://faculty.nps.edu/thuynh/conference%20proceedings%20papers/paper_29_sosece_2_huynh_paper.pdf.

IOOS, 2013. "*The U.S. Integrated Ocean Observing System: Our Eyes on Our Oceans, Coasts, and Great Lakes*". Available at: http://www.ioos.noaa.gov

JAMSHIDI M., 2009. "*System of Systems Engineering: Innovations for the 21st Century*". John Willey & Sons Inc.

JENNINGS N.R., 2001. "*An Agent-Based Approach for Building Software Complex Systems*" Communications of the ACM, 44(4).

KANG K., COHEN S., HESS J., NOVAK W., PETERSON S., 1990. "*Feature-Oriented Domain Analysis (FODA): Feasibility Study*". SEI Technical Note CMU/SEI-90-TR-21. Available at: http://resources.sei.cmu.edu/asset_files/TechnicalReport/1990_005_001_15872.pdf

KEATING C., ROGERS R., UNAL R., DRYER D., SOUSA-POZA A., SAFFORD R., PETERSON W., RABADI G., 2003, "*System of systems engineering*", Engineering Management Journal, 15(2): pp.36

KHANDPUR R., 2005. "*Printed Circuit Boards: Design, Fabrication, and Assembly*". 1st Edition, McGraw-Hill.

KOTOV V., 1997. "*System of Systems as Communicating Structures*". HP Labs technical reports. Available at: http://www.hpl.hp.com/techreports/97/HPL-97-124.pdf

KRUEGER C.W., 2006. "*New Methods in Software Product Line Development*". 10th International Software Product Line Conference, pp.95-99.

LEE J., KOTONYA G., 2010. "*Combining Service Orientation with Product Line Engineering*", IEEE Software, 27(3), pp.35-41.

LEVESON, N.G., HEIMDAHL, M.P.E., HILDRETH, H., REESE, J.D., 1994. "*Requirements Specification for Process-Control Systems*". IEEE Transactions on Software Engineering, 20(9), pp. 684-707.

LEWIS G., MORRIS E., SIMANTA S., SMITH D., 2011. "*Service Orientation and Systems of Systems*". IEEE Software, 28(1): pp. 58-63, IEEE Computer Society.

LIBELIUM, 2013. Smart Parking and environmental monitoring. Available at: http://www.libelium.com/smart_santander_smart_parking.

LUSKASIK S.J., 1998. "*Systems, system of systems, and the education of engineers*". Artificial Intelligence for Engineering Design, Analysis, and Manufacturing, 12(1): pp.55-60.

MAIER M. W., 1998. "*Architecting Principles for Systems-of-Systems*", Systems Engineering, 1(4): pp.267-284, Wiley Online Library.

McGREGOR, J. D., 2004 "*Software Product Lines*" in Journal of Object Technology (JOT 2004), 3(3): pp. 65-74. Available at: http://www.jot.fm/issues/issue_2004_03/column6.

MEYERS B.C., SMITH J.D., CAPELL P., PLACE P.R.H., 2006. "*Requirements Management in a System-of-Systems Context: A Workshop*". SEI Technical Note CMU/SEI-2006-TN-015. Available at: http://resources.sei.cmu.edu/asset_files/TechnicalNote/2006_004_001_14690.pdf

MORRIS E., LEVINE L., MEYERS C., PLACE P., PLAKOSH D., 2004. "*System of Systems Interoperability (SOSI): Final Report*". SEI Technical Note CMU/SEI-2004-TR-004. Available at: http://resources.sei.cmu.edu/asset_files/TechnicalReport/2004_005_001_14375.pdf

OCL, 2012. OMG Object Constraint Language v2.3.1. Available at: http://www.omg.org/spec/OCL/2.3.1/

RAMOS, M. A., PENTEADO, R. A. D., 2008. "*Embedded Software Revitalization through Component Mining and Software Product Line Techniques*". Journal of Universal Computer Science (J.UCS), 14(8): pp. 1207-1227. Available at: http://www.jucs.org/jucs_14_8/embedded_software_revitalization_through/jucs_14_08_1207_1227_ramos.pdf.

RAMOS, M.A., MASIERO, P.C., BRAGA, R.T.V., PENTEADO, R.A.D., 2012. "*Reengineering legacy systems towards system of systems development*". IEEE 13th International Conference on Information Reuse and Integration (IRI), pp. 624 - 630

RAMOS, M.A., MASIERO, P.C., BRAGA, R.T.V., PENTEADO, R.A.D., 2013. ”*From software product lines to system of systems: Analysis of an evolution path*”. IEEE 14th International Conference on Information Reuse and Integration (IRI), pp. 394 – 401.

ROBERTS, J., WALKER, R., 2010 “*Flying Robots to the Rescue*” IEEE Robotics & Automation Magazine, 17(4): pp. 8-10, IEEE Robotics and Automation Society.

SAGE A.P., CUPPAN C.D., 2001. “*On the Systems Engineering and Management of System of Systems and Federation of Systems*”. Journal Information-Knowledge-Systems Management, 2(4): pp.325-345, IOS Press.

SAUSER B., BOARDMAN J., GOROD A., 2009. “*System of Systems Management*”. In “*System of Systems Engineering: Innovations for the 21st Century*”, Mo Jamshidi, John Willey & Sons Inc., Chapter 8.

SEI, 2014. Software Engineering Institute, Carnegie Mellon University, USA. “*System of Systems – Guiding successful engineering, governing and acquiring of system of systems*”. Available at: http://www.sei.cmu.edu/sos

SHIBASAKI R., PEARLMAN J. S., 2009. “*System of System Engineering of GEOSS*”. In “*System of Systems Engineering: Innovations for the 21st Century*”. John Willey & Sons Inc, Chapter 22.

SMITH II J.D., 2006. “*Topics in Interoperability: Structural Programmatics in a System of Systems*”. SEI Technical Note CMU/SEI-2006-TN-037. Available at: http://resources.sei.cmu.edu/asset_files/TechnicalNote/2006_004_001_14741.pdf

SMITH II J.D., PHILLIPS D.M., 2006. “*Interoperable Acquisition for Systems of Systems: The Challenges*”. SEI Technical Note CMU/SEI-2006-TN-034. Available at: http://resources.sei.cmu.edu/asset_files/TechnicalNote/2006_004_001_14732.pdf

SOCHOS, P.; RIEBISCH, M. E PHILIPPOW I., 2006. “*The Feature-Architecture Mapping (FArM) Method for Feature-Oriented Development of Software Product Lines*”. In: 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, p. 308-318.

SUTCLIFFE A., 2003. “*Scenario-Based Requirements Engineering”*. 11th International Conference on Requirements Engineering (RE '03), Mini-tutorial. Monterey Bay, CA. Available at: http://csis.pace.edu/~marchese/CS775/Papers/sutcliffe_scenario_based.pdf

TIAN Z., WANG J., ZHU J., DING W., LIANG H.Q., 2011. "*Visualizing and Modeling Interaction Relationships Among Entities*". IBM Corp. Patent No. US 7930678 B2. USA. Available at: http://patentimages.storage.googleapis.com/pdfs/US7930678.pdf

VERIFONE®, 2013. *The way to pay*^{TM}. Available at: http:\\www.verifone.com

VON DER BEECK M., 1994. "*A comparison of Statecharts variants*". In: Proceedings of Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'94), Lecture Notes in Computer Science, (863), pp. 128–148., Springer-Verlag.

WEISS, D. M.; LAI C. T. R., 1999. "*Software Product-Line Engineering: A Family-Based Software Development Process*". Addison-Wesley, USA.

WS-ADDRESSING, 2004: *Web Services Addressing*. W3C. Available at: http://www.w3.org/Submission/ws-addressing

WS-DISCOVERY, 2009: *Web Services Dynamic Discovery*. OASIS. Available at: http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.html

WS-NOTIFICATION, 2006. *Web Services Notification*. OASIS. Available at: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn

ZHOU B., DVORYANCHIKOVA A., LOBOV A., MARTINEZ J.L. 2011. "*Modeling system of systems: A generic method based on system characteristics and interface*". 9th IEEE International Conference on Industrial Informatics, pp. 361-368.

ZIADI T., JÉZÉQUEL J-M., FONDEMENT F., 2003. "*Product line derivation with UML*", In: Proceedings of Software Variability Management Workshop, University of Groningen. Available at: http://www.irisa.fr/triskell/publis/2003/Ziadi03b.pdf