Science of Computer Programming 191 (2020) 102416

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Automated reasoning framework for traceability management of system of systems

Bedir Tekinerdogan[a,*], Ferhat Erata[b,c]

[a] Information Technology, Wageningen University, the Netherlands
[b] Department of Computer Science, Yale University, CT, USA
[c] UNIT Information Technologies, Research & Development, Turkey

A R T I C L E   I N F O

Article history:
Received 17 March 2019
Received in revised form 25 January 2020
Accepted 28 January 2020
Available online 3 February 2020

Keywords:
System of systems
Traceability
Metamodeling
Domain specific language

A B S T R A C T

An important aspect in system of systems (SoS) is the realization of the capabilities in different systems that work together. Identifying and locating these capabilities are important to orchestrate the overall activities and hereby to achieve the overall goal of the SoS. System elements and capabilities in SoS however, are rarely stable and need to evolve in different ways and different times in accordance with the changing requirements. To manage the SoS and cope with its evolution it is necessary that the dependency links to the capabilities and the system elements can be easily traced. Several approaches have been proposed to model traceability and reason about these by extending a predetermined set of possible trace links with fixed semantics. However, for the context of SoS a fixed traceability model with fixed traceability semantics is limited to consider the various different and changing scenarios. In this article, we first present the different traceability requirements for managing traceability in the context of SoS. Subsequently, we present the metamodel and the corresponding domain specific language to support modeling traceability and traceability analysis approaches within the evolving SoS context. Further, we provide the tool support for automated reasoning of traceability of SoS capabilities and system elements. We illustrate and discuss the approach for the application to a smart city SoS.

1. Introduction

Whereas traditionally systems were addressing a single domain, current systems have to be composed of multiple systems that need to be integrated in a coherent way. To be able to design, analyze, implement and maintain such large so-called systems of systems (SoS) [1], a *Systems of Systems Engineering* (SoSE) approach is required [2], [1]. Traditionally, SoSE has been heavily used in the defense domain but is now also increasingly being applied to non-defense related problems such as air and auto transportation, healthcare, global communication networks, search and rescue systems, space exploration and many other SoS application domains.

One of the key challenges of SoS besides its complexity is the continuous need for evolution whereby new system elements and capabilities are deployed, and unnecessary system elements and capabilities will be given up. The evolution of SoSs is inherently continuous, whereby adaptations will be made continuously to meet the changing requirements. Evolution is often not local but systemic in that it impacts multiple system elements and capabilities together.

* Corresponding author.
E-mail address: bedir.tekinerdogan@wur.nl (B. Tekinerdogan).

https://doi.org/10.1016/j.scico.2020.102416
0167-6423/© 2020 Elsevier B.V. All rights reserved.

To manage this size of complexity and the evolution of SoS, it is necessary that the corresponding dependency links can be easily traced. This is because changes to capabilities or system elements as such can have consequences for the overall SoS performance. Due to the dynamic behavior of SoS, traceability management within the context of SoS is not only important in the initial construction or integration of the SoS, but also in later phases in which the systems can evolve independently.

The notion of traceability is not new and much research has been carried out in different domains including requirements engineering, model-driven development, and aspect-oriented software development [3], [4]. In general, a traceability model defines the types of the trace relations, such as *depends-on*, *refines*, *satisfies*, etc. [5], [6], [7]. For ensuring traceability very often a fixed traceability model is employed by the existing tools, that define a predetermined set of possible trace relations and their corresponding semantics. For single homogeneous systems with a predetermined semantics that does not need to change, adopting a single static traceability model is usually not a serious problem. However, in the case of dealing with complex heterogeneous systems as in the case of SoSs, these approaches will be limited. In the context of SoS, it is therefore required to enable the adoption of different traceability models with their own specific semantics, and herewith the corresponding different traceability analysis approaches. This need for a richer and dynamically configurable traceability semantics further requires a platform in which system elements as well as capabilities are modeled as first class abstractions, and the traceability model can be easily adapted to support a customized traceability analysis.

In this article, we aim to address this problem of heterogeneous dynamic traceability semantics by first providing a list of requirements for addressing traceability in SoS. Based on these requirements we propose a metamodel for traceability in SoS that is mapped to a domain specific language, which can be further used to support systematic traceability analysis approaches. We present the key different traceability analysis scenarios including consistency checking of constructed SoS models, extraction and depiction of trace links, and the impact analysis of a change in the SoS elements or capabilities. The overall approach is applied to the smart city engineering case study.

The remainder of the paper is organized as follows. In section 2, we briefly describe the background on SoS engineering. In section 3 we present the case study on smart city engineering that we will use to illustrate the problem statement and the adopted approach. In section 4, we present the requirements for traceability within the context of SoS. In section 5 we present the traceability metamodel for SoS together with the tracelink semantics and the traceability analysis. Section 6 presents the domain specific language based on the traceability metamodel and the defined semantics. Section 7 presents the tool of the presented approach. In section 8 we present the related work and finally in section 9 we conclude the paper.

2. System of systems and capability engineering

Systems can be grouped together to create SoS. There are many kinds of systems, including natural systems, social systems, and technological systems. Systems that are created by and for people are referred to as Engineered Systems [2].

In an SoS, the individual system components have managerial and operational independence, whereas the overall purpose of the system is to provide a function or service that cannot be provided by the individual systems independently. In general, systems in SoSs are being employed in various combinations to provide different capabilities (Fig. 1). Different types of SoS can be distinguished based on different properties which we have extensively discussed in our earlier work [2], [8]. A recurring classification is based on central management of the systems, a common agreed purpose and the independent ownership of the systems based on which four key categories of SoSs are distinguished:

Directed SoS are built and managed to fulfill specific purposes. The SoS is centrally managed to fulfill the agreed purpose. The component systems can operate independently, but their normal operational mode is subordinated to the centrally

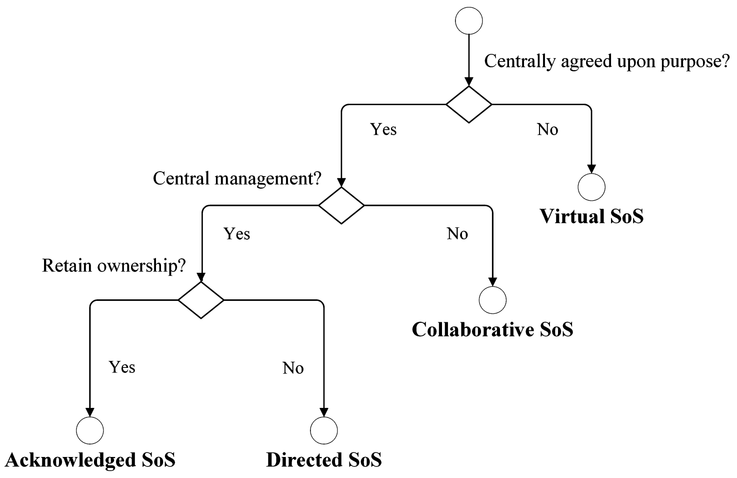


Fig. 1. Types of system of systems [2].

Table 1
Possible evolution scenarios in SoS; system element is a component, sub-system, system, or SoS.

Nr.	Description of the change scenario
S_1	System element is added to the SoS
S_2	System element is removed from the SoS
S_3	System element is updated in the SoS
S_4	A capability is added to a system element in the SoS
S_5	A capability is removed from a system element in the SoS
S_6	A capability from a system element is updated in the SoS

managed purpose. *Acknowledged SoS* have also a recognized objective and central management. In contrast to Directed SoS the constituent systems retain their independent ownership. Changes in the systems are based on collaboration between the SoS and the system. *Collaborative SoSs* have a commonly agreed purpose but do not have a central management. The component systems interact voluntarily to fulfill agreed-upon central purposes. *Virtual SoSs* lack a central management authority and a centrally agreed-upon purpose. Further, the component systems have their independent ownership. It should be noted that the classification of SoS based on governance and management criteria. We refer to [8] in which a multi-dimensional classification of SoS is provided.

Related to SoS is the term *capability* that is widely used across many SoS application domains, and which has actually taken on various specific meanings across, and even within, those domains. Terms such as capability-based acquisition, capability engineering and management, life capability management, capability sponsor, etc. are now ubiquitous in multiple application domains.

Henshaw et al. [9] indicate that different communities use the terms with similar, but distinctly different meanings, and have identified eight perspectives of capability. They define capability as the ability to do something and capability engineering as the overarching approach that links value, purpose, and solution of a systems problem. Often, at the highest granularity level, capability is usually defined as the emergent property of SoS. In our context we will adopt the broader definition of property or service of each system element, whether it is the result of emergence or not.

An SoS is rarely stable and needs to evolve continuously. The need for evolution might be due to the changing context and the changing requirements. In general, in an SoS either the system elements or the capabilities can change. System elements can be at different granularity including component, sub-system, system, or even an SoS. Based on this, Table 1 shows the possible evolution scenarios in SoS. Given the three different types of granularity of system elements that we distinguished the table thus captures $6 \times 4 = 24$ different change scenarios.

3. Case study: smart city engineering

To illustrate the need for traceability in SoS we will adopt the case study for smart city engineering [10], [11], [12], [13]. It is expected that the gross of the world population will live in urban cities in the near future. This will have a huge impact on future personal lives and mobility. A smart city uses information and communication technology (ICT) to enhance the quality and performance of urban services, to reduce costs and resource consumption, and to engage more effectively and actively with its citizens. Sectors that have been developing smart city technology include government services, transport and traffic management, water and waste, health care, and energy. Smart city systems typically can be considered as an SoS since it consists of multiple systems that are integrated to achieve a given goal. Smart city applications are developed with the goal to improve the management of urban flows and allowing for real-time responses to challenges. Based on the common aspects of smart cities as described in the literature [14] we have derived the decomposition view of a smart city as shown in Fig. 2. Each rectangle in the figure shows a particular smart system that is integrated into the overall smart city SoS. Table 2 shows the capabilities of a selected set of smart systems in the overall SoS. Note that the elements listed in Table 2 are all capabilities and need to be read as such. Some constituent systems might have similar names (e.g. Smart Traffic). The capability could be realized with such a system but in any case despite possible similar names conceptually the elements are different (system has capability).

Fig. 2 shows the static structural perspective of the smart SoS. However, like in any SoS also in smart city SoS the different system elements are interdependent and collaborate to realize their own capabilities and to create capabilities that emerge out of the collaboration. Fig. 3 thus shows a partial view of the smart city SoS including the system elements and the various behavioral dependency relations among the system elements. The rectangles represent the constituents systems while the arrows represent the dependency links.

The smart SoS is a typical example of an SoS that constantly evolves. To manage the SoS it is important to identify, model and trace these changes. We can identify a number of example scenarios that show the need for traceability management:

- *S1-Add system to the smart grid of the city*
  The smart grid of a smart city is responsible for the smart electricity supply network that uses digital communications links to regulate and optimize energy usage in the city. Adding a new system to the overall smart city SoS can require updating the procedures for allocation, regulation, and optimization procedures for the smart metering and distribution

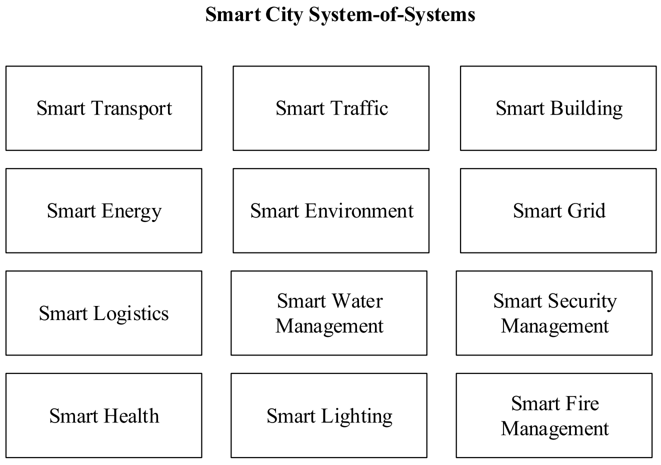


Fig. 2. Modules of smart city.

Table 2
Example capabilities of selected system elements in smart city case.

System element	Capability
Smart Transport	Smart Notification Smart Traffic Collision Avoidance Variable Speed
Smart Energy	Smart metering Smart provisioning Smart distribution Smart energy storage
Smart Logistics	Smart Inventory Management Warehouse Management Security Smart Packaging Transportation Material Handling Vehicle to Vehicle Communication
Smart Traffic	Smart Parking Smart Lighting Smart Monitoring Car to Car Communication
Smart Farming	Smart Livestock farming Variable Rate Fertilization Smart Climate Monitoring Greenhouse automation Crop Management

of energy. Based on Fig. 3 we can also derive that changes to *Smart Grid* will impact *Smart Energy* which on its turn will impact the other system elements.

- *S2-Monitor waste and pollution*
  Waste and pollution prevention and reduction is an important aspect of a city to enhance the well-being of the citizens. In general, multiple system units will need to have the capability for monitoring and controlling the waste and pollution in a city. To optimize this process, it is important that each system element is traced for this capability. In case of changes to the system elements or the capabilities, this should be explicitly notified. For the given models of smart city this means that capabilities for monitoring of waste and pollution prevention and reduction must be added to the corresponding system elements.
- *S3-Enhance security*
  Security is usually an important capability in a city that actually is related to and has an impact to multiple different system elements as well as other capabilities. For enhancing the overall security in a city it is important that we have a clear picture on the system elements that have an impact on supporting security, that is, which require the security capability. Changes to the system elements and/or the capabilities might lead to security threats and weaken

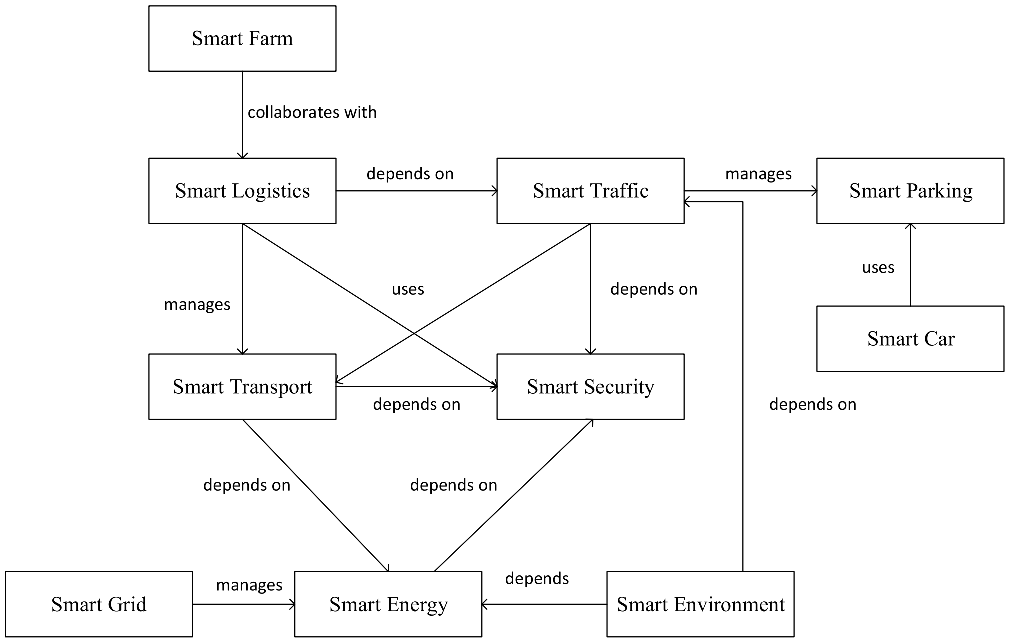


Fig. 3. Partial view of smart city with trace links among system elements.

the system. Hence, traceability is here not only needed in the initial design and analysis of the SoS, but also required in the evolution of the SoS. We will elaborate on this scenario in section 7.

- *S4-Optimize help in emergency*
  A city can cope with different emergency states that require a coordinated action of the system units. To support the effective and efficient communication and emergency handling the different capabilities related to this need to be well-defined and allocated over the different system units. Here, traceability is needed to detect the anticipated and unanticipated incidents, broadcast and communicate the incident, and realize the required support for resolving the emergency.

The above scenarios are only selected examples that could be required in an SoS, and we could easily identify several other scenarios that impact the SoS and that require traceability management. In general, traceability in SoS requires the identification of the system elements and capabilities for a given set of queries. For example, for realizing the scenario "enhance security" we need to identify all the system elements that are related to security. For the scenario "reduce pollution" we need to identify the system elements that are related to pollution. In some cases we could derive from the names of the system elements which capabilities are implemented, however, very often this is usually not that straightforward and manual traceability management is limited. Moreover, each capability might also map to more than one system element. As stated before, these scenarios and the need for traceability analysis is inherent in SoSs due to their very dynamic character. SoSs are constantly evolving in which system elements are added, removed, updated, with different capabilities that are implemented or emerged after a composition of the system elements. An explicit traceability management approach dedicated for SoS is needed to support the understanding and maintenance of the evolving SoSs.

4. Requirements for SoS traceability

Based on the literature on traceability and SoS we provide a set of requirements for traceability in SoS in the following sub-sections.

4.1. Explicit modeling of SoS structure and the capabilities

In order to explicitly reason about traceability it is necessary that the corresponding capabilities, and the system elements are explicitly modeled as first class abstractions. The detail of this required capability model could range from just a description of its name to a full semantic model including attributes such as stakeholder, the domain of the capability, the date at it was raised, the impact that it has, etc.

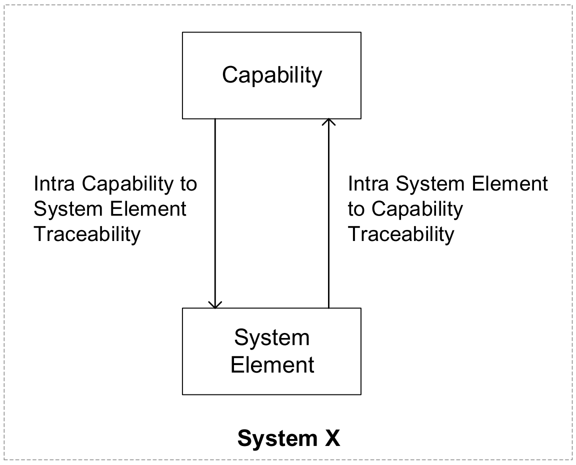

Capability
Intra Capability to System Element Traceability
Intra System Element to Capability Traceability
System Element
System X

Fig. 4. Traceability relationships within a single system.

4.2. *Explicit modeling of dependency relations in the SoS*

Within an SoS, system elements might be related to other system elements, and address capabilities. To support traceability, it is necessary to make these relations explicit. This can be only done when dependency relations are recorded as traceability links. For this, traceability links should be specified as first class abstractions in the adopted traceability model.

4.3. *Support for automated tracing queries*

The explicit models for capabilities and the traceability help to define the links between the different capabilities and the system elements. For a large-scale SoS manually tracing the changes is not tractable for the human engineer. To support the tracing, the system should provide automated support for defining generic and user-defined queries to identify and trace the capabilities. Traceability analysis can be carried out for different purposes including consistency analysis and impact analysis.

4.4. *Dynamic adaptation of traceability semantics*

To support traceability, different types of traceability models have been used with their own tools [3], [15], [16]. A traceability model defines the types of trace relations, such as depends-on, refines, satisfies, etc. [7]. For ensuring traceability, very often a fixed traceability model is employed by the existing tools, that define a predetermined set of possible trace relations and their corresponding semantics. For homogeneous single systems with a predetermined semantics that do not need to change, adopting a single static traceability model is usually not a serious problem. However, in the case of dealing with complex heterogeneous SoS, instead of a one-size-fits-all approach, it is required to enable the adoption of different traceability models with their own specific semantics, and herewith the corresponding different traceability analysis approaches. On its turn, this requires a platform in which the traceability model can be easily adapted to support a customized traceability analysis.

4.5. *Support for traceability within and across system boundaries*

An SoS typically consists of multiple integrated systems or system elements. As stated before, system elements can be at different granularity levels including component, sub-system, system, or SoS. Tracing should be supported within and across system boundaries. Obviously, capabilities will be allocated and realized in various systems of the SoS. To understand the relations among the capabilities and system elements, it is necessary to model traceability for the given system element. Fig. 4 shows the abstract model for tracing within a system. We define here two types of traceability: (1) intra capability to system element traceability (2) intra system element to capability traceability. In the first case we are interested in tracing the system elements that are related to a given capability. In the second case we are interested in tracing the capabilities that are related to a given system element.

Fig. 5 and Fig. 6 presents the conceptual model for traceability relationships across systems in an SoS. In Fig. 5 two types of relations are defined that we think are necessary. *Inter capability to capability traceability* defines the traceability of a capability in one system to another capability of another system. *Inter system element traceability* defines the traceability of a system element in one system to another system element in another system.

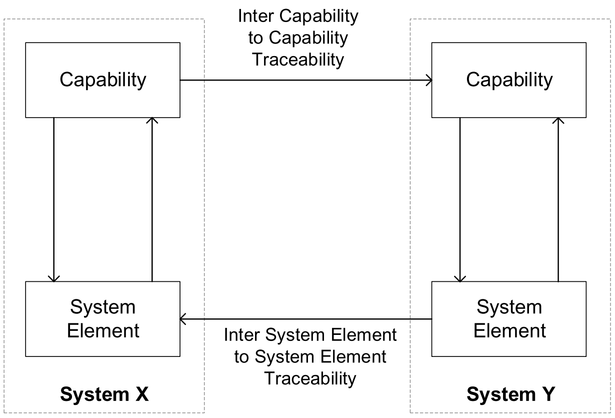

Inter Capability to Capability Traceability
Capability
Capability
System Element
System Element
Inter System Element to System Element Traceability
System X
System Y

Fig. 5. Traceability relationships across systems in SoS - capability to capability.

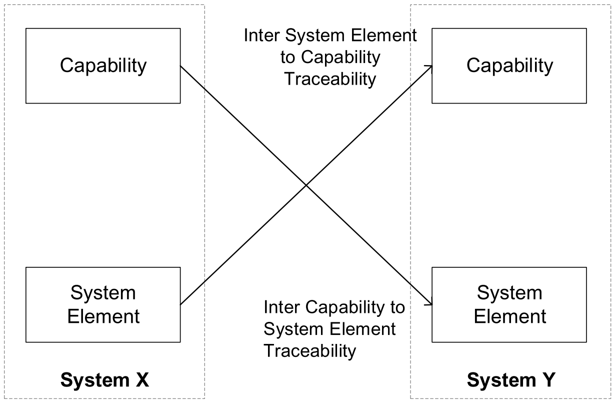

Capability
Inter System Element to Capability Traceability
Capability
System Element
Inter Capability to System Element Traceability
System Element
System X
System Y

Fig. 6. Traceability relationships across systems in SoS - capability to system elements.

Fig. 6 adopts the traceability between systems, but here the relations are defined from a capability to a system element and vice versa. It should be noted that the relations from Fig. 6 can also be derived from the traceability relations of 4 and 5. The traceability relations of Fig. 6 can be practical to optimize the tracing.

5. Metamodel for traceability in SoS

In this section we will provide the traceability metamodel, the required traceability semantics and the traceability analysis.

5.1. *Traceability metamodel*

Fig. 7 shows the traceability metamodel that we will adopt to support the traceability management of SoSs. The metamodel represents and integrates both the structural modeling of SoS using system elements and capabilities, and the required tracing modeling. The left part of the metamodel represents the traceable elements, which are *capability* or *system element*. System elements can be recursively decomposed. At the highest level resides the *SoS*, which can include two or more systems. A system on its turn can include *sub-systems* or *components*. Sub-systems can have further sub-systems or components. The right part of the metamodel represents the trace link elements. A tracelink has one or more source and target elements. A trace link has a particular scope, that is, either within a single system scope (IntraTraceLink) or across multiple system elements (InterTraceLink). A trace link further has different semantics which could be, for example, *depend*, *require*, *satisfy*, *conform*, *generate*, etc. We will elaborate on these in the following sub-section. Table 3 shows the possible set of trace link semantics that we have derived from [17]. In essence, we can identify the following types of traceability links:

- capability to system element
- capability to capability
- system element to system element

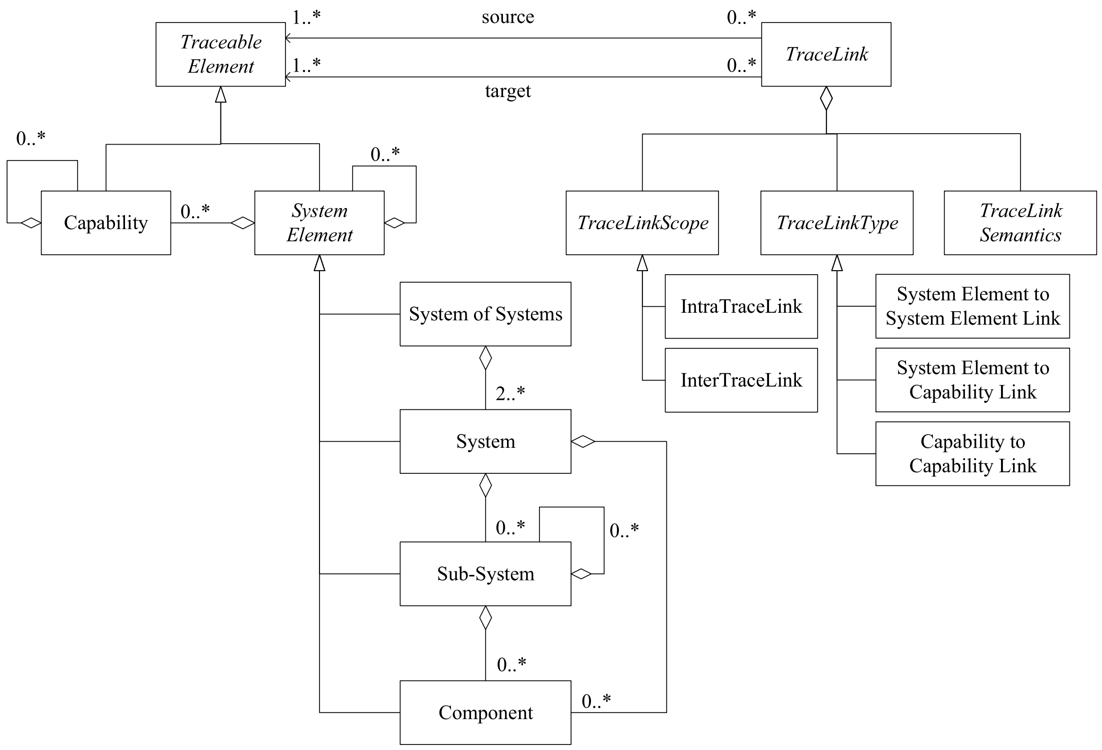


Fig. 7. Traceability metamodel.

Table 3
Semantics of trace links.

Trace-link	Trace-link semantics
	System Element → *System Element*
Depends	S_1 depends on S_2
Uses	S_1 uses S_2
Collaborates with	S_1 collaborates with S_2
Part of	S_1 is part of S_2
Refines	S_1 refines S_2
Manages	S_1 manages S_2
Reports to	S_1 reports to S_2
	System Element → *Capability* (→ *System Element*)
Implements	S_1 implements C_1
Requires	S_1 requires C_1
Owns	S_1 owns C_1
Shares with	S_1 shares C_1 with S_2
	Capability → *Capability*
Refines	C_1 refines C_2
Parts of	C_1 is part of C_2
Composed of	C_1 is composed of $C_2 \dots C_n$
Conflicts with	C_1 depends on C_2
Precedes	C_1 should be realized before C_2
Causally after	C_1 should be realized after C_2

5.2. Traceability analysis

At the level of models (SoS for city engineering) we explicitly specify the semantics using Alloy specification language.

Once we have defined the metamodel and the traceability semantics we can now focus on possible traceability analysis approaches. Traceability analysis aims to trace a capability or system element for different purposes. This could be for example, for checking the consistency or for analyzing the impact of a change of a system element or capability in the SoS. The scope of the traceability can cover a single system scope or cross the boundaries of systems. We term the first type of traceability analysis *intra-traceability analysis*, while the second type is termed as *inter-traceability analysis*.

Table 4 shows a selected set of identified queries that are relevant for traceability analysis within a system in the context of SoSs. In the table, multiple different queries have been defined that aim to make explicit the trace links between the

Table 4
Traceability analysis queries within system boundary.

	Traceability within system boundary
Q_1	What are the elements that implement capability C_1 within system S_1?
Q_2	What are the elements that refines capability C_1 within system S_1?
Q_3	What are the elements that are linked to element S_1?
Q_4	What are the elements that do not implement a capability?
Q_5	What are the capabilities that are linked to C_1?
Q_6	What is the trace-depth level of capability C_1?
Q_7	What is the trace-depth level of system element S_1?
Q_8	Does system element S_1 implement capability C_1?

Table 5
Traceability analysis queries across system boundaries.

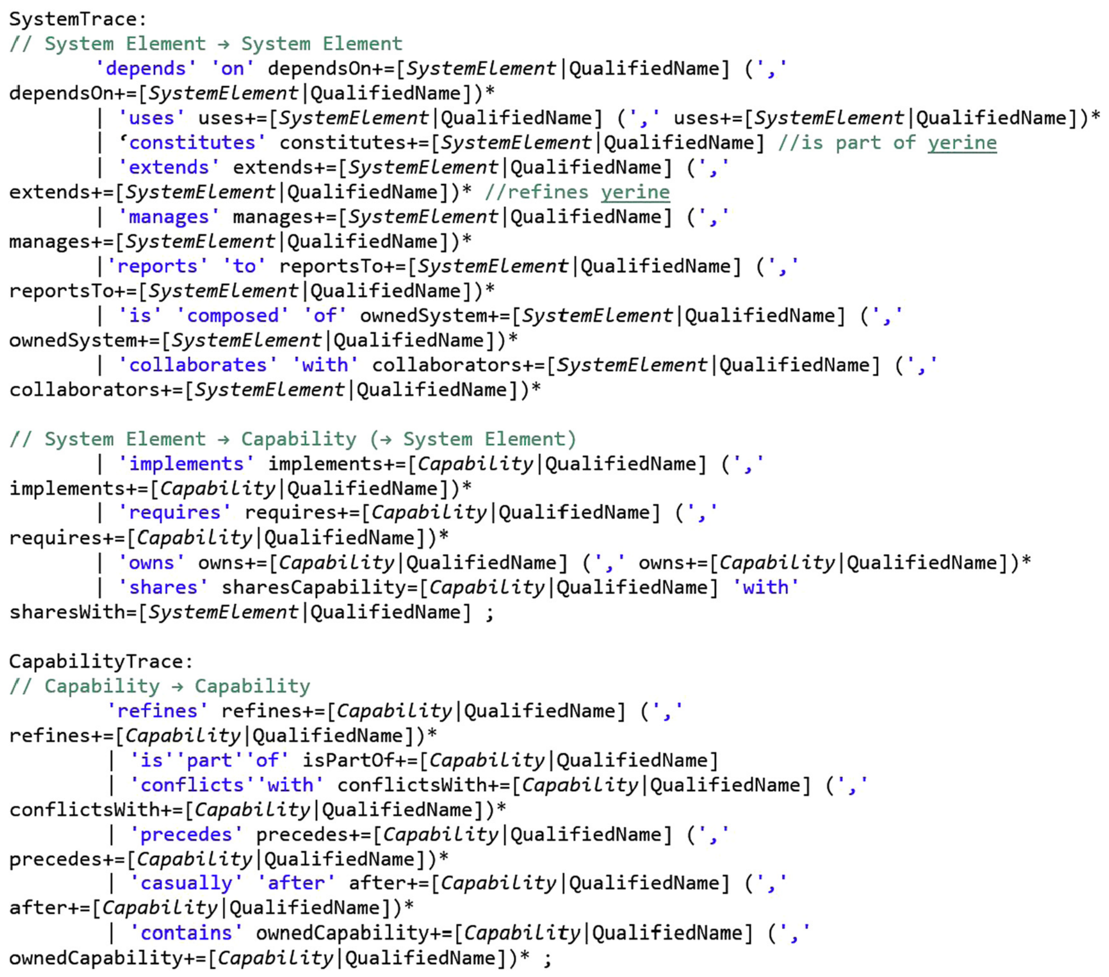
	Traceability across system boundaries
Q_9	What are the elements that implement capability C_1 within system $\{S_1, S_2 \ldots S_n\}$?
Q_{10}	What are the elements that do not implement a capability within system $\{S_1, S_2 \ldots S_n\}$?
Q_{11}	What are the capabilities that are linked to C_1 within system $\{S_1, S_2 \ldots S_n\}$?
Q_{12}	What is the trace-depth level of capability C_1 within system $\{S_1, S_2 \ldots S_n\}$?
Q_{13}	What is the trace-depth level of system element S_1 within system $\{S_1, S_2 \ldots S_n\}$?
Q_{14}	Is capability C_1 realized in the scope of $\{S_1, S_2 \ldots S_n\}$?

```
grammar eu.modelwriter.systemofsystems.ModelAndTrace with org.eclipse.xtext.common.Terminals
generate modelAndTrace "http://www.modelwriter.eu/systemofsystems/ModelAndTrace"
TraceModel: (elements+=AbstractElement)*;
AbstractElement:  System | Import;
QualifiedName: ID ('.' ID)*;

Import:'import' importedNamespace=QualifiedNameWithWildcard;

QualifiedNameWithWildcard: QualifiedName '.*'?;

System:       'System' name=QualifiedName title=STRING? (traces+=SystemTrace (','
traces+=SystemTrace)*)? (('{'
        (elements+=TraceableElement)*
        (traces+=TraceDeclaration ';')*
    '}') | ';') ;

TraceableElement: SystemElement | Capability;

SystemElement: System | Component;

TraceDeclaration: (from=[Capability|QualifiedName] traces+=CapabilityTrace (','
traces+=CapabilityTrace)*) | (from=[System|QualifiedName] traces+=SystemTrace (','
traces+=SystemTrace)*);


Capability:'Capability' name=ID title=STRING? (traces+=CapabilityTrace (','
traces+=CapabilityTrace)*)? (('{'(ownedCapabilities+=Capability)*'}') | ';') ;

Component:'Component' name=ID title=STRING? (('{'
              (ownedCapabilities+=Capability)*
        '}') | ';') ;
```

Fig. 8. Grammar DSL for system-of-systems (based on metamodel).

trace elements, that is, system elements and capabilities. These queries on their turn can be used to support the high level traceability analysis goals. For example, for analyzing the impact of a change of a capability Q in a system we might need to use the queries that can select the system elements that are related to Q, and then further check these elements for the trace-depth.

Table 5 shows a selected set of identified queries that are relevant for traceability analysis across single system boundaries in the context of SoSs. The queries are partially similar to the queries of a single system boundaries but also consider a broader scope.



```
SystemTrace:
// System Element → System Element
       'depends' 'on' dependsOn+=[SystemElement|QualifiedName] (','
dependsOn+=[SystemElement|QualifiedName])*
        | 'uses' uses+=[SystemElement|QualifiedName] (',' uses+=[SystemElement|QualifiedName])*
        | 'constitutes' constitutes+=[SystemElement|QualifiedName] //is part of yerine
        | 'extends' extends+=[SystemElement|QualifiedName] (','
extends+=[SystemElement|QualifiedName])* //refines yerine
        | 'manages' manages+=[SystemElement|QualifiedName] (','
manages+=[SystemElement|QualifiedName])*
        |'reports' 'to' reportsTo+=[SystemElement|QualifiedName] (','
reportsTo+=[SystemElement|QualifiedName])*
        | 'is' 'composed' 'of' ownedSystem+=[SystemElement|QualifiedName] (','
ownedSystem+=[SystemElement|QualifiedName])*
        | 'collaborates' 'with' collaborators+=[SystemElement|QualifiedName] (','
collaborators+=[SystemElement|QualifiedName])*

// System Element → Capability (→ System Element)
        | 'implements' implements+=[Capability|QualifiedName] (','
implements+=[Capability|QualifiedName])*
        | 'requires' requires+=[Capability|QualifiedName] (','
requires+=[Capability|QualifiedName])*
        | 'owns' owns+=[Capability|QualifiedName] (',' owns+=[Capability|QualifiedName])*
        | 'shares' sharesCapability=[Capability|QualifiedName] 'with'
sharesWith=[SystemElement|QualifiedName] ;

CapabilityTrace:
// Capability → Capability
        'refines' refines+=[Capability|QualifiedName] (','
refines+=[Capability|QualifiedName])*
         | 'is''part''of' isPartOf+=[Capability|QualifiedName]
         | 'conflicts''with' conflictsWith+=[Capability|QualifiedName] (','
conflictsWith+=[Capability|QualifiedName])*
         | 'precedes' precedes+=[Capability|QualifiedName] (','
precedes+=[Capability|QualifiedName])*
         | 'casually' 'after' after+=[Capability|QualifiedName] (','
after+=[Capability|QualifiedName])*
         | 'contains' ownedCapability+=[Capability|QualifiedName] (','
ownedCapability+=[Capability|QualifiedName])* ;
```

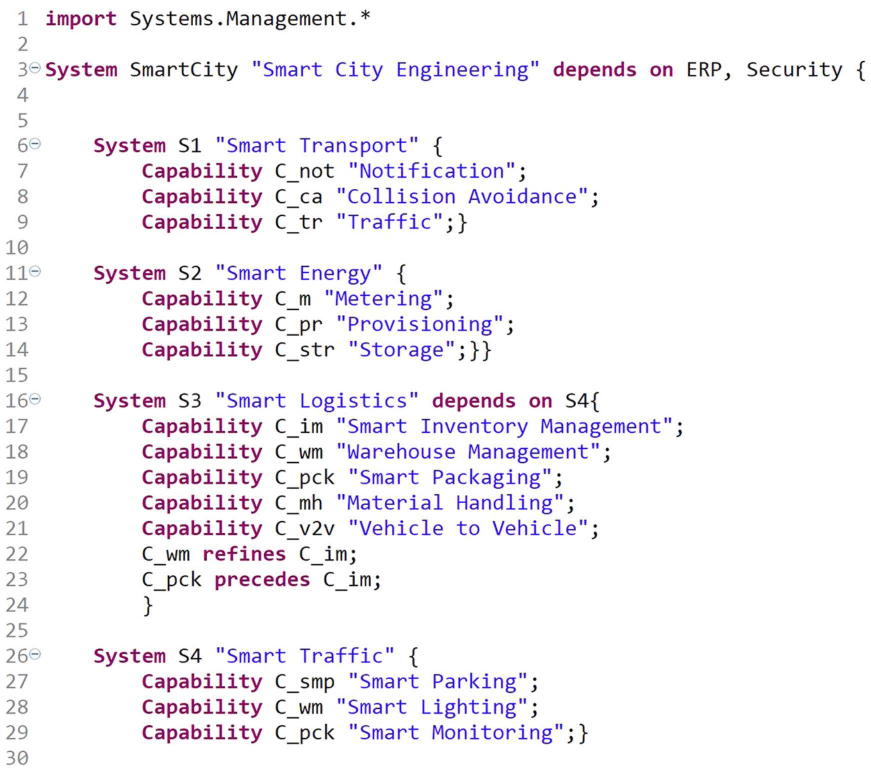
Fig. 9. Grammar DSL for system-of-systems (based on metamodel).

6. Trace-SoS: DSL for the SoS traceability metamodel

In the previous section we have described the metamodel for traceability in SoS. To support the modeling of the SoS, and the traceability of the system elements and capabilities, we have developed a domain specific language (DSL) called *Trace-SoS* based on the corresponding metamodel. A DSL is a tailor-made language for a specific problem domain [18], [19] that provides the abstractions of that domain. Several benefits can be provided for the development and use of DSLs including increasing the development productivity, supporting the communication among domain experts, supporting platform independence, supporting code generation, and a more elaborate and precise validation. To develop the DSL we have applied the method as defined by Stembeck and Zdun [18]. For developing the DSL framework, we have used the Eclipse framework including the Xtext Language Generator tools. Xtext Language Generator is used to generate textual domain specific language.

Our focus is on the domain of SoS and traceability which were combined in a single metamodel. The abstractions for the DSL consist thus of the concepts as defined in the metamodel. In principle, the DSL consists of three basic parts (1) SoS modeling (2) traceability modeling, and (3) trace semantics modeling. Fig. 8 shows the grammar of the part of the SoS traceability metamodel that is used for defining the structure of the SoS. Fig. 9 shows the part of the DSL that is related to the definition of the trace links. All the earlier defined trace link types have been implemented in the DSL and thus can be used to model an SoS with the different trace links. Fig. 10 shows an example DSL instance for the smart city system that was discussed before.

For the evaluation of the DSL we have looked at several theoretical studies in the literature that can be used to assess novel DSLs. We applied the Framework for Qualitative Assessment of DSLs (FQAD) which has been recently published [20]. The framework is based on the ISO/IEC 25010:2011 standard and defines a set of quality characteristics for evaluating a DSL including: Functional suitability, Usability, Reliability, Maintainability, Productivity, Extensibility, Compatibility, Expressiveness, Reusability, and Integrability.



```
import Systems.Management.*

System SmartCity "Smart City Engineering" depends on ERP, Security {


    System S1 "Smart Transport" {
        Capability C_not "Notification";
        Capability C_ca "Collision Avoidance";
        Capability C_tr "Traffic";}

    System S2 "Smart Energy" {
        Capability C_m "Metering";
        Capability C_pr "Provisioning";
        Capability C_str "Storage";}}

    System S3 "Smart Logistics" depends on S4{
        Capability C_im "Smart Inventory Management";
        Capability C_wm "Warehouse Management";
        Capability C_pck "Smart Packaging";
        Capability C_mh "Material Handling";
        Capability C_v2v "Vehicle to Vehicle";
        C_wm refines C_im;
        C_pck precedes C_im;
        }

    System S4 "Smart Traffic" {
        Capability C_smp "Smart Parking";
        Capability C_wm "Smart Lighting";
        Capability C_pck "Smart Monitoring";}

```

Fig. 10. DSL instance of the DSL for system-of-systems.

Functional suitability refers to the degree to which a DSL is fully developed and likewise includes all the necessary functionality. We have focused on two key issues including SoS modeling and traceability modeling. Both aspects have been thoroughly analyzed from the literature and the required functionality has been mapped to the metamodel and the DSL.

Usability of a DSL is the degree to which a DSL can be used by specified users to achieve specified goals. We can state that the DSL is relatively simple but expressive and can be relatively easily learned. We have also explicitly focused on addressing the different concerns which we believe is helpful.

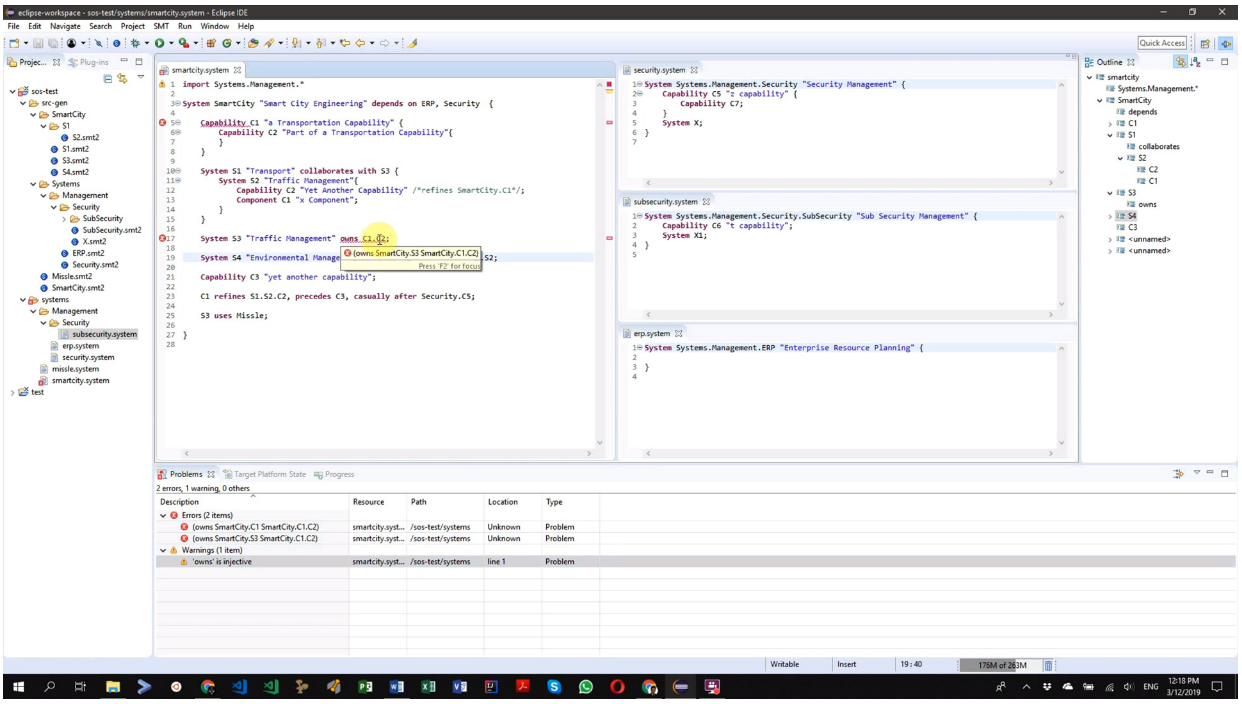
Reliability of a DSL is defined as the property of a language that aids producing reliable programs. The provided DSL has been developed using the Eclipse environment. The language has an editor with the default editing features including auto-complete. The language is defined according to the well-defined principles and it has precise semantics. The Eclipse editor provides all the required instruments for debugging and handling code errors. Extensibility defines the degree to which a language has general mechanisms for users to add new features. For adding new features to the languages, the language developer toolset can be used and the language can be easily extended. This is of course also the benefit of the Eclipse and the related tools that we used as a language workbench.

Integrability defines how easily a DSL can be integrated with other languages and modeling tools. We have developed the language framework as a plug-in within the Eclipse framework. The Eclipse platform allows the developer to extend Eclipse applications like the Eclipse IDE with additional functionalities via plug-ins. Integration of the language with other languages can be done using the Eclipse IDE and the overall OSGI component model.

Maintainability defines the degree to which a language is easy to maintain. The DSL can be altered and new concepts and concept extensions can be added. The DSL has been carefully designed regarding the separation of concerns principle. This has led to a modular structure of the overall language framework as well as within the DSL. Productivity of a DSL refers to the degree to which a language promotes programming productivity. Productivity is a characteristic related to the amount of resources expended by the user to achieve specified goals. Productivity can be increased because of the adoption of high level domain-specific language concepts. Compatibility defines the degree to which a DSL is compatible to the domain and development process. The language framework that we have provided is agnostic to the development process.

Reusability is defined as the degree to which a language constructs can be used in more than one language. The specifications that are defined in the DSL can be reused in the overall environment.

Expressiveness is defined as the degree to which a problem solving strategy can be mapped into a program naturally. In other words, expressiveness is the relation between the program and what the programmer has in mind. Hereby it is important that there is a one-to-one correspondence between concepts and their representation in the language. Further the right abstraction level must be selected so as not to use too generic or too specific concepts. The DSL that has been developed is based on a thorough domain analysis whereby we have modeled each concept in the corresponding metamod-


smartcity.system

```
import Systems.Management.*

System SmartCity "Smart City Engineering" depends on ERP, Security  {

    Capability C1 "a Transportation Capability" {
        Capability C2 "Part of a Transportation Capability"{
        }
    }

    System S1 "Transport" collaborates with S3 {
        System S2 "Traffic Management"{
            Capability C2 "Yet Another Capability" /*refines SmartCity.C1*/;
            Component C1 "x Component";
        }
    }

    System S3 "Traffic Management" owns C1.C2;

    System S4 "Environmental Manage                                  S2;

    Capability C3 "yet another capability";

    C1 refines S1.S2.C2, precedes C3, casually after Security.C5;

    S3 uses Missle;

}
```

(owns SmartCity.S3 SmartCity.C1.C2)
Press 'F2' for focus

security.system

```
System Systems.Management.Security "Security Management" {
    Capability C5 "z capability" {
        Capability C7;
    }
    System X;
}
```

subsecurity.system

```
System Systems.Management.Security.SubSecurity "Sub Security Management" {
    Capability C6 "t capability";
    System X1;
}
```

erp.system

```
System Systems.Management.ERP "Enterprise Resource Planning" {

}
```

Problems

2 errors, 1 warning, 0 others

Description	Resource	Path	Location	Type
Errors (2 items)				
(owns SmartCity.C1 SmartCity.C1.C2)	smartcity.syst...	/sos-test/systems	Unknown	Problem
(owns SmartCity.S3 SmartCity.C1.C2)	smartcity.syst...	/sos-test/systems	Unknown	Problem
Warnings (1 item)				
'owns' is injective	smartcity.syst...	/sos-test/systems	line 1	Problem

Fig. 11. Snapshot of the tool Trace-SoS.

els of the languages. We can state that there is indeed a one-to-one correspondence between the identified concepts and the representations in the language. The abstraction level of the elements has been carefully defined to be usable for the developer. The language elements are expressive enough to specify the problem, while too specific details are handled by traceability analysis programs.

7. Tool support and traceability management process

In this section we describe the tool Trace-SoS that adopts the DSL of the traceability SoS metamodel of Fig. 7. A snapshot of the tool is shown in Fig. 11. The left part of the tool shows the explorer and SoS view in which we the structure of the SoS can be modeled. The view on the right shows the trace view that depicts all the defined tracelinks. A corresponding Youtube video has been prepared to illustrate the usage of the tool (see: https://youtu.be/lRYoc8HZ-sQ and https://youtu.be/KQYHd7XLHMg).

The tool can be used to model an SoS and thereby also the tracelinks to support traceability management. For traceability management we have also implemented the queries of Table 3. Using the queries we can realize traceability management and provide the results for the following scenarios (1) SoS modeling and consistency checking (2) traceability link extraction and description (3) impact analysis (4) dynamic changing of trace semantics.

The process for defining and realizing traceability management queries is shown in Fig. 12. The left part of the figure shows the process for modeling the system decomposition consisting of SoS, systems, sub-systems, and components. After modeling the system decomposition, the capabilities and the corresponding trace links and their semantics are defined. An example of such a system is shown in the decomposition view of the smart city case study as shown in Fig. 2 and the uses view as shown in Fig. 3, and the example capability set as shown in Table 2. These elements are all expressed using the DSL as described in the previous section, and shown in the snapshot of the tools as shown in Fig. 11. Using the DSL in the tool we can thus easily develop the structure of an SoS with the required system elements, capabilities, and trace links.

While the left part of the process model includes the preparation, the right part of the process defines the usage of the tool and the developed models to support traceability management queries. For this, first a higher level query will be defined. An example of such a query is for example: *"What is the impact of changes to smart logistics to the security in the overall smart city?"*. These higher level (business) queries are translated to the queries that we have defined in Table 3.

For this it is important to have a sufficient knowledge on the addressed problem, in this case, security. As stated earlier in corresponding the scenario (S3), security is usually an important capability in a city that actually is related to and has an impact to multiple different system elements as well as other capabilities. For enhancing the overall security in a city it is important that we have a clear picture on the system elements that have an impact on supporting security, that is, which require the security capability. Changes to the system elements and/or the capabilities might lead to security threats and weaken the system. Security is often related to the protection of personnel, hardware, software, networks and data

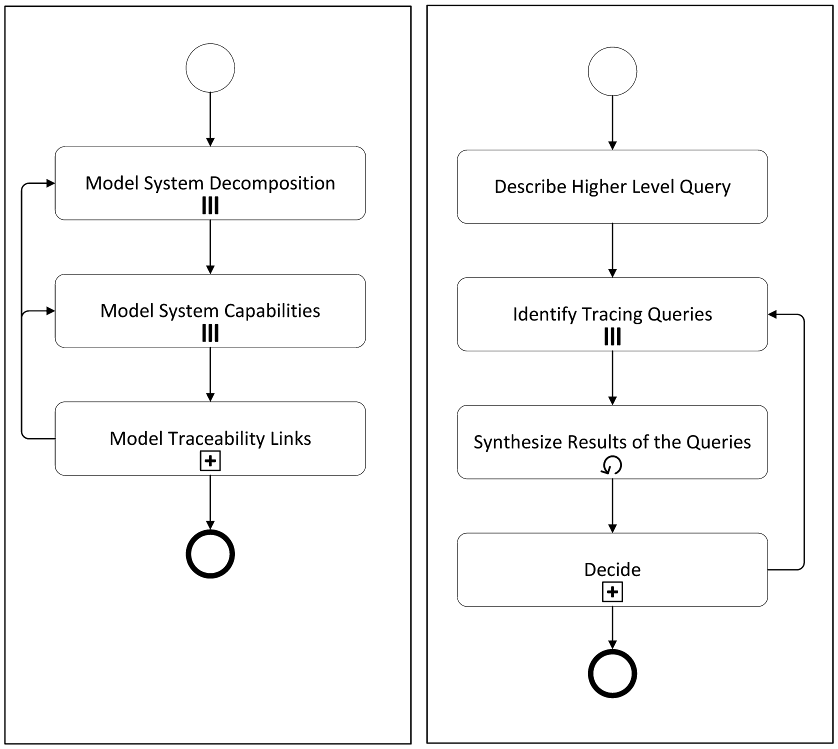

Model System Decomposition
Model System Capabilities
Model Traceability Links
Describe Higher Level Query
Identify Tracing Queries
Synthesize Results of the Queries
Decide

Fig. 12. Process for realizing traceability management queries.

from actions and events that could cause serious loss or damage, Two key capabilities can be identified: *access control* and *surveillance*. Access control is needed to block potential attackers. Surveillance is needed to monitor possible anomalies and breaches and likewise take action. We could elaborate on this further but it should be clear that the required capabilities for the system elements need to be defined properly to ensure system wide security. Given this scenario we would thus need the higher level business query into the following refined queries from Table 4:

What are the elements that implement access control capability within smart city SoS?
What are the elements that implement monitoring capability within smart city SoS?
What are the capabilities that are linked to access control within smart city SoS?
What are the capabilities that are linked to monitoring within smart city SoS?
Is capability access control realized in the scope of $S1, S2, \ldots, Sn$*?*

These queries can be defined and executed using the tool. A synthesis of the results of the queries can then be used to support the further decision making process.

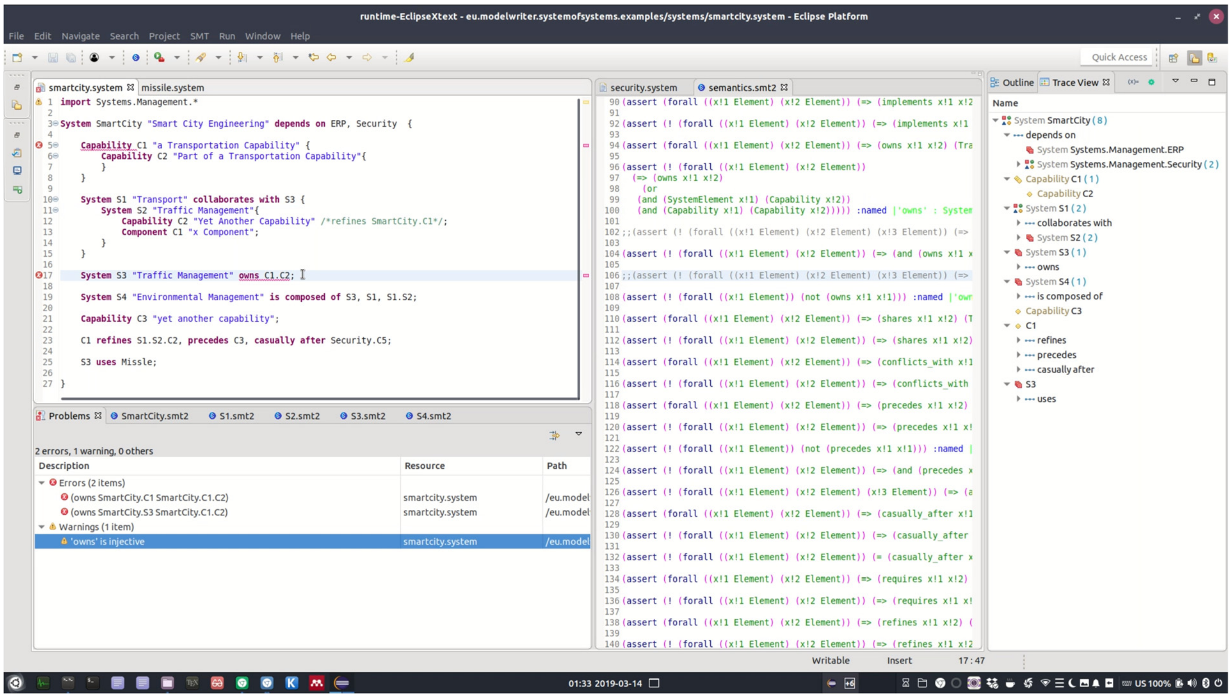
One of the key scenarios in SoS modeling is also whether the defined structure is consistent. After the SoS has been modeled we can automatically check whether the developed structure is consistent with respect to the defined rules. In addition, based on the defined trace links, new trace links can be automatically inferred and contradictions among traces can be detected (see Fig. 13). The tool takes the defined SoS model and their manually assigned traces as input, and automatically deduces, the missing trace links.

For identifying and depicting the trace links we can use the earlier defined query language which implements the query types of Table 3. The details about the configuration and extraction of tracability links and semantics can be found on the following Youtube video: https://youtu.be/3cP2wY0oLXA.

8. Related work

In fact, much has been written about traceability and we could identify many different studies that tackle different aspects of traceability in these domains. The IEEE provides the following definition of traceability [21]: “Traceability is the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship one another; for example, the degree to which the requirements and design of a given software component match.”

In requirements engineering lots of work has been done on tracing requirements from the stakeholders and in the design process [22]. In the model-driven engineering approach traceability is considered important for tracing model elements [23]. The problem of traceability has also been addressed by the AOSD community encompassing the adoption of aspects



```
import Systems.Management.*

System SmartCity "Smart City Engineering" depends on ERP, Security  {

    Capability C1 "a Transportation Capability" {
        Capability C2 "Part of a Transportation Capability"{
        }
    }

    System S1 "Transport" collaborates with S3 {
        System S2 "Traffic Management"{
            Capability C2 "Yet Another Capability" /*refines SmartCity.C1*/;
            Component C1 "x Component";
        }
    }

    System S3 "Traffic Management" owns C1.C2;

    System S4 "Environmental Management" is composed of S3, S1, S1.S2;

    Capability C3 "yet another capability";

    C1 refines S1.S2.C2, precedes C3, casually after Security.C5;

    S3 uses Missle;

}
```

```
2 errors, 1 warning, 0 others
Description                                   Resource           Path
▾ Errors (2 items)
    (owns SmartCity.C1 SmartCity.C1.C2)       smartcity.system   /eu.model
    (owns SmartCity.S3 SmartCity.C1.C2)       smartcity.system   /eu.model
▾ Warnings (1 item)
    'owns' is injective                       smartcity.system   /eu.model
```

```
(assert (forall ((x!1 Element) (x!2 Element)) (=> (implements x!1 x!2

(assert (! (forall ((x!1 Element) (x!2 Element)) (=> (implements x!1

(assert (forall ((x!1 Element) (x!2 Element)) (=> (owns x!1 x!2) (Tra

(assert (! (forall ((x!1 Element) (x!2 Element))
  (=> (owns x!1 x!2)
    (or
   (and (SystemElement x!1) (Capability x!2))
   (and (Capability x!1) (Capability x!2))))) :named |'owns' : System

;;(assert (! (forall ((x!1 Element) (x!2 Element) (x!3 Element)) (=>

(assert (! (forall ((x!1 Element) (x!2 Element)) (=> (and (owns x!1 x

;;(assert (! (forall ((x!1 Element) (x!2 Element) (x!3 Element)) (=>

(assert (! (forall ((x!1 Element)) (not (owns x!1 x!1))) :named |'owr

(assert (forall ((x!1 Element) (x!2 Element)) (=> (shares x!1 x!2) (T

(assert (! (forall ((x!1 Element) (x!2 Element)) (=> (shares x!1 x!2)

(assert (forall ((x!1 Element) (x!2 Element)) (=> (conflicts_with x!1

(assert (! (forall ((x!1 Element) (x!2 Element)) (=> (conflicts_with

(assert (forall ((x!1 Element) (x!2 Element)) (=> (precedes x!1 x!2)

(assert (! (forall ((x!1 Element) (x!2 Element)) (=> (precedes x!1 x!

(assert (! (forall ((x!1 Element)) (not (precedes x!1 x!1))) :named |

(assert (! (forall ((x!1 Element) (x!2 Element)) (=> (and (precedes x

(assert (! (forall ((x!1 Element) (x!2 Element) (x!3 Element)) (=> (a

(assert (forall ((x!1 Element) (x!2 Element)) (=> (casually_after x!1

(assert (! (forall ((x!1 Element) (x!2 Element)) (=> (casually_after

(assert (! (forall ((x!1 Element) (x!2 Element)) (= (casually_after x

(assert (forall ((x!1 Element) (x!2 Element)) (=> (requires x!1 x!2)

(assert (! (forall ((x!1 Element) (x!2 Element)) (=> (requires x!1 x!

(assert (forall ((x!1 Element) (x!2 Element)) (=> (refines x!1 x!2) (

(assert (! (forall ((x!1 Element) (x!2 Element)) (=> (refines x!1 x!2
```

```
Name
▾ System SmartCity (8)
  ▾ depends on
      System Systems.Management.ERP
    ▸ System Systems.Management.Security (2)
  ▾ Capability C1 (1)
      Capability C2
  ▾ System S1 (2)
    ▸ collaborates with
    ▸ System S2 (2)
  ▾ System S3 (1)
    ▸ owns
  ▾ System S4 (1)
    ▸ is composed of
    Capability C3
  ▾ C1
    ▸ refines
    ▸ precedes
    ▸ casually after
  ▾ S3
    ▸ uses
```

Fig. 13. Snapshot of the tool trace-SoS showing traceability configuration and consistency.

Table 6
Traceability focus in different domains.

Domain	Traceability focus
Requirements Engineering	Tracing requirements elements within requirements Tracing to and from design artefacts
Architecture Design	Tracing within and across architecture views Tracing from and to requirements Tracing to design and implementation artefacts
Model-Driven Development	Tracing (generated) model elements
Aspect-Oriented Software Development	Tracing crosscutting concerns
System of Systems	Tracing within system elements and across system elements Tracing capabilities

throughout the lifecycle [24], [25]. In our earlier work we have focused on modeling traceability within architecture views and for different lifecycle activities [3], [26].

Traceability in SoS is a more recent approach that builds on the approaches in these above domains. Our study can be considered from this perspective and is complementary to these approaches. Tracing of capabilities can benefit from the work of requirements engineering, since capabilities could be, from one perspective, be considered as higher level requirements.

Traceability from the architecture domain can be used to support the traceability among system elements. In particular, the notion of views and traceability within architecture and across architecture views can be used to support SoS traceability. Our metamodel and the related DSL could be further enhanced to cope also with architecture view traceability.

Traceability of crosscutting capabilities can benefit from aspect-oriented software development. In an SoS the capabilities will be typically distributed and allocated over different system elements. Some capabilities will have a more systemic character cannot be localized in one system unit and be scattered over different system units. We could state that some capabilities will have a crosscutting property [24], [27]. Aspect-Oriented Software Development provides solution for the crosscutting problem and could as such be used to solve the problem of crosscutting capability problems. In the context of this article this would require enhancing the DSL for expressing queries related to crosscutting concerns.

A summary of the focus on traceability is provided in Table 6. In this article we have elaborated on our earlier work but focused on SoS and did not explicitly consider the notion of views or lifecycle activities such as tracing to SoS requirements, SoS design artefacts, etc. This is however an open research issue in the context of SoS and we consider this as part of our future work.

9. Conclusion

Traceability is an important quality factor that has been addressed in various domains to improve other quality factors such as understandability, maintenance and adaptability. In this paper, we have built on the general literature on traceability and explored traceability within the context of system of systems (SoS). For managing SoS it is important to provide a proper traceability management that can identify the system elements and the capabilities. An important property of SoS is the dynamic behavior and the heterogeneity which complicates the traceability management. We have provided the key requirements for traceability in SoS and based on these developed a metamodel for traceability in SoS that integrates both SoS modeling and tracelink modeling. To address the required rich semantics of SoS we have also provided the relevant trace link semantics. If needed these trace link semantics can be easily extended. To provide automated support for traceability management we have developed the corresponding DSL and integrated this in our tool Trace-SoS that has been particularly developed for traceability management in SoS. With the tool, SoS structure together with the capabilities and the tracelinks can be easily modeled. The tool further provides the mechanisms for checking the consistency of the modeled SoS, extract and depict the tracelinks, and support impact analysis of a change of a system element and/or capability. We believe that the current tool can already be applied for traceability management of SoS. We have illustrated the tool for the modeling and traceability analysis of smart city SoS.

We have discussed the well-known classification of SoSs in the literature including directed SoS, acknowledged SoS, Collaborative SoS, and Virtual SoS. Our approach and the tool can be adopted for all these types of SoS. However, modeling traceability in directed SoS will be the easiest thanks to a strong central management and the possibility to design the system for traceability earlier on. Virtual SoS does not have a central management, no agreed purpose and the systems in the SoS are largely independent. This makes it difficult to integrate traceability analysis in such systems. Our future work will elaborate on adopting the tool method for these different types of SoSs in different application domains. In particular traceability management for non-directed SoSs will be challenging. In addition, we will also further experiment with the DSL and the tool and enhance this if necessary. Eventually we aim to apply it to more industrial case studies to enhance both the state-of-the-art of system of systems and traceability management in general.

References

[1] J.S. Dahmann, K.J. Baldwin, Understanding the current state of us defense systems of systems and the implications for systems engineering, in: 2008 2nd Annual IEEE Systems Conference, 2008, pp. 1–7.
[2] B. Tekinerdogan, Engineering Connected Intelligence: A Socio-Technical Perspective, Wageningen University, Wageningen, 2017.
[3] B. Tekinerdogan, C. Hofmann, M. Akşit, J. Bakker, Metamodel for tracing concerns across the life cycle, in: Early Aspects: Current Challenges and Future Directions, 2007, pp. 175–194.
[4] B. Tekinerdogan, F. Scholten, C. Hofmann, M. Aksit, Concern-oriented analysis and refactoring of software architectures using dependency structure matrices, in: Proceedings of the 15th Workshop on Early Aspects, 2009, pp. 13–18.
[5] F. Erata, M. Challenger, B. Tekinerdogan, A. Monceaux, E. Tüzün, G. Kardas, Tarski: a platform for automated analysis of dynamically configurable traceability semantics, in: Proceedings of the Symposium on Applied Computing, 2017, pp. 1607–1614.
[6] F. Erata, A. Goknil, B. Tekinerdogan, G. Kardas, A tool for automated reasoning about traces based on configurable formal semantics, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 959–963.
[7] F. Erata, C. Gardent, B. Gyawali, A. Shimorina, Y. Lussaud, B. Tekinerdogan, G. Kardas, A. Monceaux, ModelWriter: text and model-synchronized document engineering platform, in: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, 2017, pp. 907–912.
[8] B. Tekinerdogan, Multi-dimensional classification of system-of-systems, in: Proc. of 14th Annual Conference System of Systems Engineering, SoSE 2019, Anchorage, Alaska, USA, May 19-22, 2019, 2019, pp. 278–283.
[9] M. Henshaw, P. Lister, A.D. Harding, D. Kemp, A.J. Daw, A. Farncombe, M. Touchin, Capability engineering - an analysis of perspectives, in: INCOSE International Symposium, vol. 21, 2011, pp. 712–727.
[10] A. Cocchia, Smart and digital city: a systematic literature review, in: Smart City, Springer, 2014, pp. 13–43.
[11] M. Lacinák, J. Ristvej, Smart city, safety and security, Proc. Eng. 192 (2017) 522–527.
[12] T.-h. Kim, C. Ramos, S. Mohammed, Smart city and IoT, Future Gener. Comput. Syst. 76 (2017) 159–162.
[13] E. Cavalcante, N. Cacho, F. Lopes, T. Batista, F. Oquendo, Thinking smart cities as systems-of-systems: a perspective study, in: Proceedings of the 2nd International Workshop on Smart, 2016, p. 9.
[14] C. Lim, K.-J. Kim, P.P. Maglio, Smart cities with big data: reference models, challenges, and considerations, Cities 82 (2018) 86–99.
[15] J. Cleland-Huang, O. Gotel, A. Zisman, et al., Software and Systems Traceability, vol. 2, Springer, 2012.
[16] J. Cleland-Huang, O.C. Gotel, J. Huffman Hayes, P. Mäder, A. Zisman, Software traceability: trends and future directions, in: Proceedings of the on Future of Software Engineering, 2014, pp. 55–69.
[17] A. Krokhin, P. Jeavons, P. Jonsson, Reasoning about temporal relations: the tractable subalgebras of Allen's interval algebra, J. ACM 50 (2003) 591–640.
[18] M. Strembeck, U. Zdun, An approach for the systematic development of domain-specific languages, Softw. Pract. Exp. 39 (2009) 1253–1292.
[19] B. Tekinerdogan, E. Arkin, ParDSL: a domain-specific language framework for supporting deployment of parallel algorithms, Softw. Syst. Model. (2018) 1–29.
[20] G. Kahraman, S. Bilgen, A framework for qualitative assessment of domain-specific languages, Softw. Syst. Model. 14 (2015) 1505–1526.
[21] IEEE Std. 610.12-1990 – IEEE standard glossary of software engineering terminology, IEEE Computer Society, Los Alamitos, CA, 1990, 169.
[22] B. Ramesh, M. Jarke, Toward reference models for requirements traceability, IEEE Trans. Softw. Eng. 27 (2001) 58–93.
[23] L. Bondé, P. Boulet, J.-L. Dekeyser, Traceability and interoperability at different levels of abstraction in model-driven engineering, in: Applications of Specification and Design Languages for SoCs, Springer, 2006, pp. 263–276.
[24] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M.P. Alarcon, J. Bakker, B. Tekinerdogan, S. Clarke, A. Jackson, Survey of analysis and design approaches, AOSD-Europe Report D 11, 2005, 8.

[25] B. Tekinerdogan, ASAAM: aspectual software architecture analysis method, in: Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture, WICSA 2004, 2004, pp. 5–14.
[26] B. Tekinerdogan, C. Hofmann, M. Aksit, Modeling traceability of concerns in architectural views, in: Proceedings of the 10th International Workshop on Aspect-Oriented Modeling, 2007, pp. 49–56.
[27] B. Tekinerdogan, C. Hofmann, M. Aksit, Modeling traceability of concerns for synchronizing architectural views, J. Object Technol. 6 (2007) 7–25.