A Systems of Systems perspective on the Internet of Things

Invited paper†
Johan Lukkien (j.j.lukkien@tue.nl)
Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, the Netherlands

The Internet of Things (IoT) refers to extending the reach of the Internet into the physical world. The realization of IoT applications involves the integrated operation of many subsystems that retain their private function. This makes IoT application deployment and integration a Systems of Systems (SoS) problem. In this paper we collect SoS properties and characteristics from the literature in order to understand common integration problems in IoT better, for which we use two running examples. We show that in particular for safety critical systems there must be means to compute and predict integrated behavior based on specifications at interfaces. We give a general coordination architecture that supports this.

***Index Terms*—Systems of Systems, Internet of Things, Smart Cities, System Engineering**

I. Introduction

Systems-of-Systems (SoS) is a term for systems that are composed of independent (autonomous) subsystems which are full-blown systems by themselves in every way. While there is growing awareness of the importance of SoS there is no clear agreement about the architectural principles guiding the design of SoS nor about the process of engineering them.

Thinking in terms of SoS brings a radical change in viewpoint. In traditional embedded systems design[1] we focus on how to effectively design and integrate subsystems to implement the functionality of an overall system. This integrated system plays a role in the physical world. With the introduction of embedded networking these subsystems are increasingly integrated through networks. In this way they form the nodes in a larger whole. This integration becomes an SoS problem when these subsystems have both a private purpose and serve the overall goal of the combined system. SoS research is about how to design, engineer, maintain and evolve a composition of subsystems while acknowledging the fact that these subsystems remain independent, are serving their own functions, and have their own management and lifecycles.

The *Internet of Things* (IoT) refers to the vision of connecting every 'thing' (to the Internet) using a unified protocol and naming scheme thus extending the reach of the Internet into the physical world while scaling up its size to tens of billions of devices. In addition, IoT changes the types of applications, because of this access to deeply embedded sensing and control. IoT is currently at the top of the 'hype cycle' with larger companies and standardization bodies in the networking domain providing standards and frameworks.

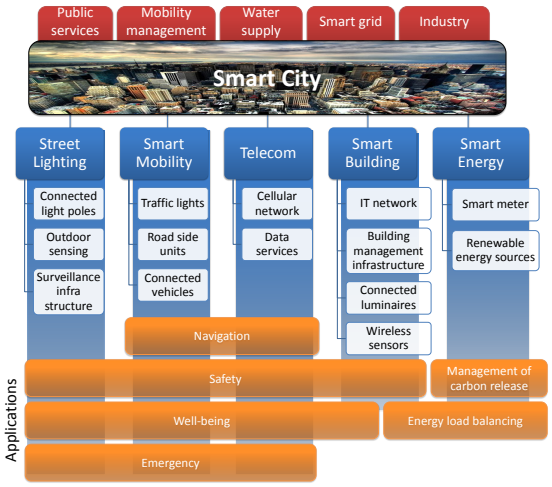

Fig. 1. *Urban (sub)systems in a Smart City can collaborate for reasons of optimization or making new applications. Applications spanning several subsystems are given in orange.*

In this paper we discuss what SoS is about based on a review of existing literature which stems mainly from the first decade of this century. We look in particular at characteristics, at architecting principles and styles, and at engineering methodologies for SoS. We relate this to IoT, showing that IoT systems follow SoS paradigms and face similar challenges. With this we aim at a renewed insight in the engineering of IoT systems.

Original examples of SoS were taken from the military domain [13] as well as transportation and avionics. A more current example concerns Smart Cities, with separate subsystems for traffic management, energy management, building management and streetlighting (to mention a few), depicted in Fig.1. New applications or optimization scenarios combine functionality of several urban systems while the regular functionality of these systems remains with its requirements of dependability and sometimes timeliness. In this paper we use Smart Cities as running example as well as a country-wide Intelligent Transportation System (ITS), depicted in Fig.2. In this figure we observe three different domains. a) The

 The research leading to this paper was partially funded by the European Unions Seventh Framework Program (FP7/2007-2013) for CRYSTAL Critical System Engineering Acceleration Joint Undertaking under grant agreement No 332830.

[1] An embedded system is by definition a subsystem in itself as it is part of a larger whole.

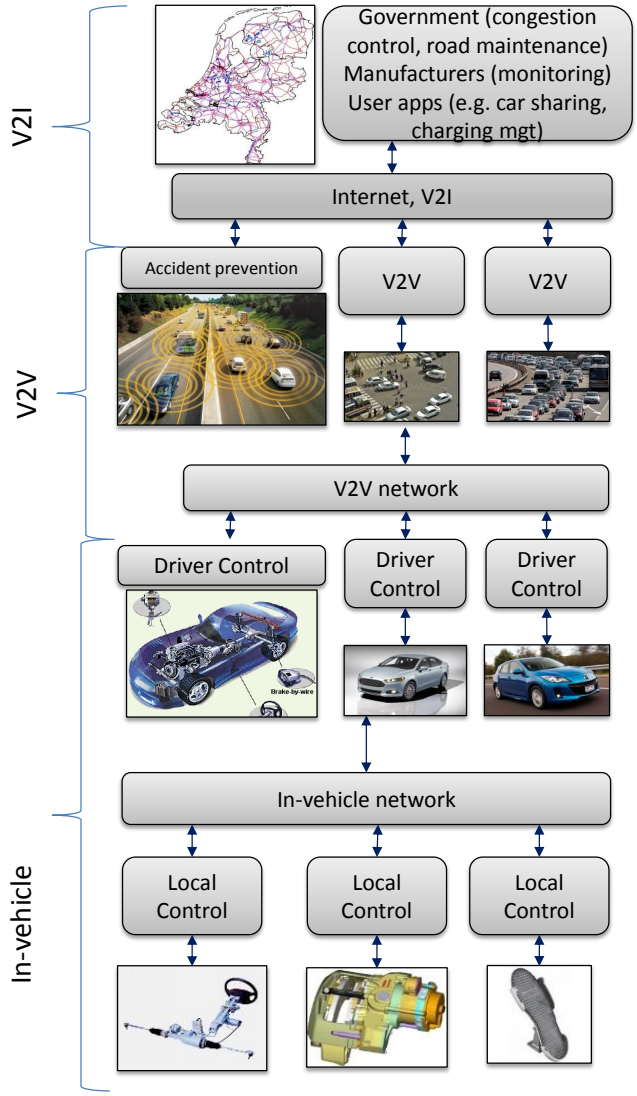

Fig. 2. *A modern Intelligent Transportation System, integrating vehicles as independent subsystems into the whole of the ITS. Applications like traffic management operate on collected data from vehicles. Besides the managerial applications, more consumer like applications are possible as well, e.g. car sharing, ride sharing, energy optimization and so on. Further explanation in the text. Subpictures, courtesy of the Internet.*

vehicle, composed of subsystems integrated in the local control system of the vehicle. Integration is through the in-vehicle network and applications concern the operation of the vehicle. Subsystems have no other function than serving the goals of the vehicle and the integration is done in the classical way after hierarchical decomposition. b) The vehicle-to-vehicle (V2V) domain with applications involving several vehicles (and posibly local infra structure) through communication, e.g., accident prevention, parking spot finding or automatic driving in a traffic jam. c) At the more global level of the country, applications are based on Vehicle to Infra structure (V2I) data collection and include traffic management or road maintenance, but also user apps like car sharing.

In this paper we first compare SoS to *Monolithic Systems* in order to understand what new aspects SoS brings. Then we examine aspects of the engineering of SoS after which we discuss consequences and solution directions. The contribution of the paper is to develop terminology for reasoning about common problems and to point towards a generic solution direction.

II. Systems of Systems and Monolithic Systems

The work of Maier[12] is one of the earlier systematic discussions on SoS. Maier introduces some properties intrinsic to an SoS: *operational independence* (subsystems have an autonomous behavior, goal and useful existence), *managerial independence* (subsystems are managed by different authorities) and *evolutionary independence* (subsystems evolve independently). In addition, *geographic distribution* is often a characteristic as well as exibiting *emergent behavior*. According to Fisher[3], geographic distribution supports the three independency properties while not being a necessary condition, and emergent behavior is the result of the subsystems having the three independency properties. DeLaurentis[4] adds to the characteristics *heterogeneity* of subsystems, *networks* as the predominant means of connecting subsystems and *trans-domain collaboration* (the need for different disciplines to collaborate, e.g., engineering, economy, policy makers). Our two examples show these aspects clearly. The Smart City example is really about integrating systems that are traditionally MLs. Independent operation, evolution and management are obvious properties, as well as heterogeneity and geographic distribution. For the ITS it is a bit more subtle as vehicles have similar characteristics, but vehicle brands have their own lifecycles and each vehicle is within the managerial domain of its owner. This ownership brings also a debate on the role of manufacturers when they collect data from the vehicles.

SoS is clearly about the design and engineering problems of integrating existing systems into a larger whole that yields new functionality not available through any of the constituent systems. These problems comprise the architectural principles of such composition, the engineering process and the technical solutions, in particular with respect to extra-functional properties and integrity of the composed systems. We examine some concepts that refer to this integration of (sub)systems to understand the difference between ML and SoS.

A. System Integration

System integration in the context of MLs refers to the concept of synthesizing (independently developed) subsystems. Typically, the specification of these subsystems follows from a decomposition of an original design of the ML, which is a well-established engineering practice. The focus lies here on interface definition, on integration methodology (e.g. vertical integration: integration of different abstraction levels, and horizontal integration: integration at similar abstraction levels), on managing and reducing dependencies (coupling) and on maximizing cohesion. The goal is to obtain a single integrated system in which the subsystems are there for serving the combined goal.

Within SoS there is also a need to integrate subsystems but there are important differences. Fisher[3] describes some implicit assumptions of system integration that are not true for SoS, viz. that the architecture is frozen in an earlier stage of the design, that the control flows and data flows are known, and that requirements as well as properties of subsystems are known upon integration. For SoS, the requirements for subsystems are not specified in a hierarchical manner; the function of each subsystem is defined by its original purpose as well as its own internal context, data, processes, etc. We call this *uncorrelated requirements* of subsystems. Second, in case of SoS there are two types of control flow: the control flow for the original purpose and the control flow coming from the SoS. We call this aspect *competition of control*. Third, within SoS there is no a priori architectural principle that guides the design of all subsystems; instead, subsystems have their own architectural principles. We call this *architectural diversity*.

Applying this to the example of Smart Cities, it is clear that the described aspects are important there. In order to understand the competition of control a scenario is required, which can be one in which an emergency vehicle influences the traffic control system to obtain a free road, as well as the telecommunication system in order to warn people on the road in advance. The competition is then with the original control in these two systems. For the ITS, the competition comes, for example, in a potentially dangerous situation where automated control takes over based on V2V communication.

In both cases, regarding this as an SoS integration problem avoids the direction of making evermore complex software while adding new functionality. Instead the focus moves to interfaces that make both functionality and control of extra-functional properties available.

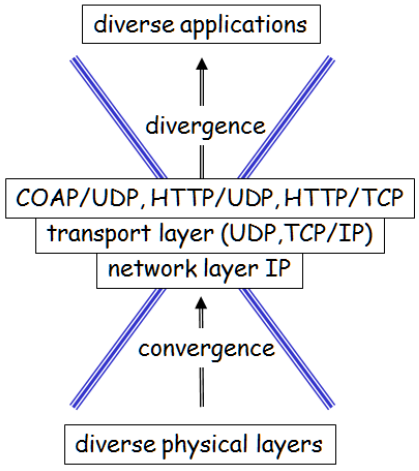

Fig. 3. *A primary interoperability is derived from being able to exchange data packets regardless of the physical networking technology used. This is achieved by the convergence towards IP transport protocols. On top of this, however, applications vary as well as the meaning of the exchanged data. From this point on, description of the semantics are required to enable interoperation for SoS.*

B. Interoperation

Interoperation refers to cooperative interactions between two or more partners to achieve objectives. These objectives can be shared (e.g., manage the traffic in an area), but can also be private to each partner and can be as simple as obeying actively some policy. For example, in a client-server interaction, the server achieves its goal by serving clients.

Premise to interoperability is the ability to communicate, which in networks is addressed in the convergence until the transport layer in the OSI stack, as depicted in Fig.3. Currently, a standardization on application protocols is observed, based on the RESTful style. On top of that meaningful information is exchanged and interpreted between interfaces, but with a divergence in semantics depending on the particular application. Interoperation requires understanding on three aspects which can be seen as stages in the interoperation: understanding, first, on how to reach the interface (discovering the interface), second, on how to perform the interoperation (understanding the interface), and third, on how the interaction contributes to the objectives (understanding the semantics). In addition, interoperation requires a certain level of trust between cooperating partners.

Aspects of interoperation are defined in several domains with slight differences, but always following this main line of reasoning. Within ML these three aspects can be entirely contained within the design, e.g. through embedding of implicit or explicit assumptions in implementations, or through protocol standardization. More recent works on component based systems and distributed systems introduce concepts that allow late binding like service discovery, service descriptions (e.g. within UPnP[8] and the Service Oriented Architectural Style (SOA, see e.g. [14]). Standards in semantic-level descriptions are RDF and OWL.

Besides standardization, interoperability within SoS need to be based on a high-level description of goals and of services since the architectural diversity implies that no assumption can be made about the inner workings of subsystems. This is called *semantic interoperability*. This means that such late binding techniques needs to be further investigated and developed for SoS. The separation between service and implementation needs to be emphasized even further, in particular using rich service interfaces that include semantic descriptions and that expose extra-functional information as well (see also below).

Looking at the example of Smart Cities, this semantic interoperability takes place at access points to the subsystem. Within ITS, standardization admits V2V communication, but for large scale data recording, semantic interoperability describing the meaning of data is more natural such as to avoid complex datastructure standardization efforts.

C. Emergence

Emergent properties refer to properties exhibited by the system as a whole that cannot be attributed to any of its subsystems in isolation[11]. Examples are extra-functional properties like latency and throughput as well as security, reliability and availability, which typically arise from system behavior over time. Hence, also the term emergent *behavior* is

Concept	ML	ML assumptions	SoS
Subsystem integration	• Horizontal and vertical integration • Interface and function definitions based on design & decomposition • Reduce coupling, maximize cohesion • Subsystems have no private goal	• Architecture frozen in early design stage • Known (and controlled) data and control flow • Requirements and properties of subsystems known upon integration	• *Uncorrelated requirements:* – Functions of subsystems defined by original, independent purpose – Subsystem behavior defined by internal processes and state • *Competition of control:* competing control flows from SoS integration and own function of subsystem • *Architectural diversity:* – Each subsystem has its own architectural principles – Fully independent lifecycles
Interoperation: • discover interface • understand and use interface functions • understand interface semantics	• Contained in design (code) • Standardization • Late binding, standard descriptions (RDF, OWL, SOA)	Embedded in code: • Knowledge about interface semantics • Knowledge about particular technologies	• High-level description of goals and services (semantic interoperability) • Rich service interfaces, including extra-functional properties • Extend late binding techniques • Negotiation
Emergence	• Addressed within the architecture • Weak	Emergent properties are addressed in (de)composition	• Make behavioral properties explicit at subsystem boundaries (rich interfaces) • Weak (whitebox), strong (blackbox) • Directed control and selforganization

TABLE I
Summary of properties and comparison between Monolithic Systems and SoS

often used. While in an ML emergent properties are typically addressed within the architecture giving them a place in the process of hierarchical decomposition, within SoS these properties require explicit attention at subsystem boundaries. In line with the discussion on interoperability, these properties must be managed at subsystem interfaces.

Because of the characteristics of SoS (e.g. independent evolution) there is an intrinsic uncertainty about the effect of operations, about the effect of failures etc. This must be taken into account at subsystem boundaries, in particular, by adopting failures, unpredictable behavior and conflicts of control as the natural mode of operation rather than as the exception.

For SoS, emergent functionality is identified as a defining property. Such emergent functionality is achieved through interoperation. Since we cannot expect to have direct and detailed control within a subsystem, required emergent functionality must be the result of policy specification at subsystem boundaries or of standardization. The emergence can furthermore be the result of directed control (see below), or from self-organization.

Chalmers[5] discriminates weak and strong emergent behavior based on whether the behavior can be deduced from the subsystems. Within SoS the emergent behavior is typically weak when assuming a white box situation in which all subsystems are transparent. However, since this is most often not the case, the emergent behavior is perceived as strong when it cannot be computed from the behavior specification at interfaces.

Emergent behavior in our example of ITS concerns V2V applications, which include cooperative driving, warnings and prevention. Cooperative driving, like platooning, is a complex control problem, which shows that such emergence is a complex matter. In order to establish such behavior, resource reservations and guarantees at interfaces are required.

Table I gives an overview.

III. Systems of Systems Engineering

A. Classification

Maier[12] discerns three types of SoS: virtual, directed, and collaborative. A directed SoS looks mostly like an ML with a centralized control. It means, in fact, that the restriction of managerial independence is dropped. The distinction remains that subsystems can also function autonomously. In a collaborative SoS the centralized control cannot enforce cooperation. Instead, applications rely on the voluntary collaboration of subsystems. In a virtual SoS there is no central control. It lacks a central agreement process upon purpose; this just emerges from the constituent systems. In all three cases, but especially in the last two, a signaling type of interaction ("commands") is not the right mode; instead, interoperation is based on negotiation. In a directed SoS this negotiation entails obtaining and exercising control in a manner similar to supervisory control, subsequently orchestrating the behavior of the subsystems. For a collaborative SoS the negotiation entails the decision on a shared goal while for a virtual SoS the decision on a shared goal is very localized.

Our Smart City example is most likely a directed SoS. The ITS example is at least collaborative but also has characteristics of a virtual SoS, especially for the safety applications that have a very localized goal. With a decrease of explicit control, emergence becomes more important. For predictability, specification of budgets at interfaces, standardization and certified behavior will be vital. By using policies at interfaces we separate the (policy) negotiation from interactions.

B. Engineering

The research reported in the SoS domain is mainly of a reflective nature: researchers and practitioners recognize that

the problems they encounter go beyond traditional system design and integration. In order to increase understanding they have generalized and subsequently taxonomized the concepts and the problems, as summarized above. Keating et al.[2] discuss SoS engineering (SoSE) in combination with systems engineering and identify a number of differences. They explain that an SoSE process must address design issues differently from traditional systems engineering in view of the given characteristics, and address system evaluation and evolution as well as system transformation for SoS DSL-based code generation is a technique that supports this.

This work seems to regard adaptation of code as part of the integration. However, independent evolution interferes with this. Because of this independent evolution, Integration of SoS is expected in a late stage of the life cycle, e.g. during or after deployment, except, possibly, for a directed SoS. This also advocates the network as the integration point.

Lewis et al.[9] describe SoSE as addressing a double challenge, viz., of generating responses extremely flexibly in changing situations while collaborating effectively across system boundaries. They define an abstract lifecycle addressing the SoS software development consisting of three steps:

1) The independent subsystems contribute a pool of software elements.
2) SoS engineers search through this pool for elements to build integrated SoS capabilities.
3) Subsequently, the relation between the SoS and the original subsystem needs to be established. The nature of this relation defines the dependencies between subsystem and the SoS. For example, if the SoS application requires access to certain data, the subsystem must allow this access, which has to be aligned with the local security policies.

This lifecycle appears to approach the design problem from a similar perspective as Component-Based Software Engineering, i.e., by composing functionality from existing components. The resulting problems are resolved during or after this composition by (re)configuration. Not all problems can be solved in this way. For example, performance problems and control conflicts might need refactoring. In addition, similar to the previous case the approach does not take into account the fact that subsystem lifecycles have become truly independent. This independence would cause repeated invocation of the SoSE engineering process upon changes in any of the constituent subsystems. Instead, we need to develop an SoSE process in which functionality of a subsystem is represented by services using rich interfaces that admit management of extra-functional properties. Such a specification will be an explicit requirement on the further evolution of subsystems. DSLs can be very useful here as they can be used to describe these interfaces.

In spite of this, the state-of-the-art for MLs is relevant to take into account when considering SoS engineering. The role of reference designs and architectures (e.g. the time-triggered architecture[10], reference architectures for IoT[6]), integration verification and tools for that, application of standards for connectivity and middleware (e.g. AUTOSAR), ever increasing use of simulators, software synthesis/code generation, and

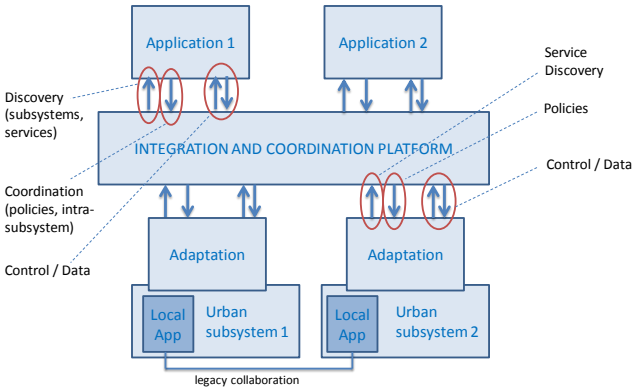

Fig. 4. *Proposed architecture for the integration of urban systems. The subsystems connect to a coordination layer using interfaces that give controlled access to the resources of the subsystem. Service discovery and policy negotiation form the basis for subsequent control and data flow. By adding an application developers API the layer has a north and a south API. Developers need similar access to develop distributed applications using service discovery and policy establishment, but now across subsystems.*

verification tools is evident and increasing. Rather new and very useful for SoS are fast integration techniques like smart adapters, and system-level awareness of control of operations. These techniques are only partly available.

The engineering of applications in the Smart City example really calls for an intermediate layer as described in Fig.4 that avoids a pairwise interaction with an explosion of dependencies. The design of collaborative applications in the V2V domain is based on standardization, extensive analysis and certification because of the safety criticality nature. It is an example where the interactions can be designed and foreseen in advance at the expense of reduced operational independence. Aspects of legacy and evolution must still be taken into account. Collaborative applications in the V2I domain might follow a similar structure as in Fig.4, separating application development from the data collection systems that are vendor or fleet owner specific.

C. Architectural principles and styles

The essential ingredients of SoS follow from the independence of the subsystems. The architecting principles must therefore address this independence. Operational and managerial independence require negotiations at interfaces to obtain access and budgets. Evolutionary independence requires loose coupling techniques and binding based on introspection and rich descriptions. Relevant styles include the following. The Service Oriented Architectural style provides explicit interface specification, separates implementation from functionality and avoids hidden state. Publish & Subscribe styles separate cooperating parties both in time and space. The RESTful style simplifies the interaction between parties by removing state.

SoSE calls for fault tolerance techniques taking deviating behavior as the norm, leading to reflective systems that examine interactions and determine the health of themselves[7]. Explicit reasoning of the system about its own health, comparing its state with expectations is a relatively new field, closely related to adaptive systems and scenario detection

mechanisms. Machine learning and anomaly detection have been around for some time, but only very application-specific.

In the literature, four architectural aspects are further emphasized. The first is that SoS is based on communication and networking. The required flexibility and decoupling from operating systems and languages, the robustness for device failures and the likes are naturally achieved by placing system boundaries at networks. Although it might be debatable, this seems a good first choice.

Second, the concept of SOAs is mentioned as a relevant style since it decouples clearly the concepts of service, implementation and specification. The current uses of SOA, however, might not be adequate as it often assumes shared ontologies and centralized discovery mechanisms that may not be appropriate across systems. In addition, there is no clear and integrated general concept of negotiation. The use of services and explicit specifications admits dynamic integration using model-based adapter technology, where formal methods support compliancy with specifications on the basis of model descriptions.

Third, some authors mention that in order for systems to evolve further, stable intermediate subsystems are required. Hence, in order to build larger systems with more functionality we have to leave the subsystems closed, and build on top of their interfaces. Leaving a system closed means really not interfering with its private architecture, operation, management, evolution, installation etc., but it also means that the subsystem is essentially self-contained in terms of self-adaptation and management. This includes join-and-leave scenarios that describe what happens with the system if constituent subsystems are joining and leaving at runtime.

Fourth, the following may be considered as a general principle: any addition or change to the SoS must at least retain existing functionality and quality within the subsystems. Differently phrased, changes must not make things worse, or the optimist variant: changes make things better.

Applying this to the example of ITS, vehicles will continuously monitor the situation around them in order to observe discrepancies between their own state and the state reported by devices around them. Also, internal monitoring procedures will be in place in order to evaluate their own healthiness. Interaction will be stateless, if possible, each vehicle listening to the vehicles in range. For the Smart City example the integration is best done using the SOA style as also indicated in Fig.4.

IV. Discussion

We have discussed Systems of Systems integration by examining the literature in relation to two examples: Smart Cities and Intelligent Transportation Systems. We see the latter as an IoT application with collaborative scenarios at two different levels: V2V and V2I. A Smart City can also be regarded as an IoT system when we consider embedded sensing and actuating as being available for integrated scenarios. The discussion led to an increased understanding of the engineering problems around SoS in general and IoT in more depth. It gives us terminology to explain phenomena that we see occurring in practice.

The ITS is also a typical example of *Cyber Physical Systems* (CPS). CPS result from the advances in ICT when applied to real-time embedded systems. While in IoT concerns of scalability and management are dominant, in CPS predictability and dependability are the drivers. In the context of SoS this is particular challenging since guarantees are required across independent domains. We see that these CPS and IoT now grow together, with dependability concerns reaching IoT systems and scalability and management concerns reaching CPS. We discuss three aspects in some more detail.

First, the managerial independence in IoT is particularly important with respect to data. Data generated by smart phones, in-home equipment, vehicles or any other data source is easily transported to a location in the cloud. This means that the data leaves the domain of the data owner (DO) to a service provider (SP). This SP in turn is subject to rules made up by public authorities (PA). For example, the PA may require the SP to give access to the data without DO being involved in any way. Proposals to deal with this include changing the reach of DO by having a private store under control of DO in which each SP stores its data of DO. Other solutions include policies and country restrictions for SP about data handling and storage.

Second, SoS increasingly have an aspect of inclusion of third parties for the development of applications. For example, in ITS third parties want to develop new safety applications or applications on top of cloud-collected data from vehicles. Examples include energy/charging optimization and vehicle sharing. Similarly, development of Smart City applications would better be done by third parties. This leads to a design in which there is an API for developers on the one hand and an API for subsystems on the other side. The developed applications lead to configuration and behavior of the subsystems through a coordination layer. Typically this is referred to as north (for application programmers) and south (for subsystems) APIs respectively, see e.g., Software Defined Networking[1]. The situation is not uncommon to an Operating Systems API that manages resources that attach to it via plug and play technology.

Third, as we have explained, subsystems provide access by exposing functionality and resources and admitting reservations at interfaces. These rich interfaces are made available to application builders and include discovery of services inside the subsystem as well as control and data flows. Combining these ideas yields a diagram as in Fig. 4 for the case of Urban Systems.

V. Conclusion

In this paper we discussed System of Systems development and integration characteristics and discussed their impact on two examples relating to the Internet of Things: Smart Cities and Intelligent Transportation Systems. The network was found to be the natural point of integration. The main characteristics of SoS, managerial, operational and evolutionary independence are relevant for these examples. We introduced three more aspects that characterize SoS integration, viz., architectural diversity, competition of control and uncorrelated

requirements. The analysis provided terminology and a recognition of common issues and complications in IoT systems. Our discussion yields a generic integration architecture based on rich interfaces that admit reservation, access control and the computation of extra-functional properties of compositions.

VI. Acknowledgements

I want to thank Pieter Cuijpers and Tanir Ozcelebi for their help in preparing the paper. I thank Louis Almeida for stimulating discussions around the topics of this paper. The research leading to this paper was partially funded by the European Unions Seventh Framework Program (FP7/2007-2013) for CRYSTAL Critical System Engineering Acceleration Joint Undertaking under grant agreement No 332830.

References

[1] X.Nguyen K.Obraczka B.A.A.Nunes, M.Mendonca. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16, 2014.

[2] R.Unal D.Dryer C.Keating, R.Rogers. Systems of systems engineering. *Engineering Management Journal*, 15, 2003.

[3] D.A.Fisher. An emergent perspective on interoperation in systems of systems, TECHNICAL REPORT CMU/SEI-2006-TR-003, ESC-TR-2006-003, 2008.

[4] D.DeLaurentis. Understanding transportation as a system of systems design problem. In *43rd AIAA Aerospace Sciences Meeting, Reno, Nevada*, AIAA-2005-0123, 2005.

[5] D.J.Chalmers. Varieties of emergence. In *The Re-Emergence of Emergence: The Emergentist Hypothesis from Science to Religion*. Oxford University Press, 2006.

[6] Bassi et al. *Enabling Things to Talk, Designing IoT solutions with the IoT Architectural Reference Model*. Springer Open, 2013.

[7] J.J.Lukkien E.U.Warriach, T.Ozcelebi. Fault-prevention in smart environments for dependable applications. In *IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems*, pages 183–184, 2014.

[8] UPnP Forum. Upnp device architecture 2.0, 2015.

[9] P.Place S.Simanta D.Smits L.Wrage G.Lewis, E.Morris. Engineering systems of systems. In *IEEE International Systems Conference*, Montreal, 2008.

[10] G.Bauer H.Kopetz. The time-triggered architecture. *Proceedings of the IEEE*, 91:112–126, 2003.

[11] L.Steels. Towards a theory of emergent functionality. In *Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 4452–461, 1991.

[12] M.W.Maier. Architecting principles for system of systems. *Systems Engineering*, 1(4):267–284, 1998.

[13] Office of the Deputy Under Secretary of Defense for Acquisition and Technology. Systems and software engineering. systems engineering guide for systems of systems, version 1.0, Washinton, DC: ODUSD(A&T)SSE, 2008.

[14] Th.Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice-Hall PTR, 2005.