Ingram C, Payne R, Perry S, Holt J, Hansen FO, Couto LD.
Modelling Patterns for Systems of Systems Architectures.
In: International Systems Conference (SysCon2014).
2014, Ottawa, Canada: IEEE.

DOI link to paper:

http://dx.doi.org/10.1109/SysCon.2014.6819249

Date deposited:

02/07/2014

Modelling Patterns for Systems of Systems Architectures

Claire Ingram, Richard Payne
Newcastle University
Newcastle upon Tyne, UK
firstname.lastname@ncl.ac.uk

Simon Perry, Jon Holt
Atego, UK
firstname.lastname@atego.com

Finn Overgaard Hansen, Luís Diogo Couto
Department of Engineering,
Aarhus University, Denmark
foh@iha.dk, ldc@eng.au.dk

***Abstract*—This paper presents an initial report on modelling patterns and architectures for system of systems (SoSs) and their constituent systems (CSs). Fundamental architectural principles for systems and SoSs and relevant work published so far are discussed and summarised. We introduce an initial set of five architectural patterns suitable for SoS design, illustrating each pattern with an SoS example and identifying how it meet some basic SoS aims. Finally, we summarise our plans for developing these ideas in the future.**

I. Introduction

The concept of an architecture is fundamental to any systems engineering undertaking, and the same principle can be extended to systems of systems (SoS) engineering. The 'Systems Engineering Guide for Systems of Systems' published by the Department of Defense, for example, considers that 'the design of an SoS consists of the architecture of the SoS together with changes to the designs of the constituent systems that enable them to work together according to the architecture.' [1]. Whilst patterns and architectural styles have been explored thoroughly for single system engineering, little comparative work so far has been published on styles and patterns applicable for SoS.

There is not yet a widely accepted single definition for SoS. The most well-known [2] describes five properties associated with SoSs: operational and managerial independence; evolutionary development; emergence (the SoS performs functions that do not reside in any individual constituent system); and distribution. The general concepts of independence, evolution and emergence have been quite widely adopted, and variations of these properties feature in subsequent definitions of SoS (e.g., Fisher [3], Boardman & Sauser [4] and Baldwin & Sauser [5]) The latter two also emphasise the diversity typically seen between constituent systems (CSs) within an SoS and that that a CS must accept the need to adapt. Abbott [6] emphasises the changeable architectural features of an SoS, suggesting that an SoS is 'open at the top' (there is no 'top level' system, new CSs may be added continually) and 'open at the bottom' ('the lowest level of a system of systems may be changed out from under it at any time'). Cocks [7] argues that an SoS '...contains one or more systems for which significant aspects of the integration and life cycle development of the component system(s) are beyond the managerial control or influence of the larger system.' For our work, we assume that for the SoS architect, the independence, challenging environment, lack of overall control, continued evolution, distribution and emergent behaviour are the key (although not the only) characteristics that an SoS architecture must support.

Architectural reuse 'allows projects to quickly identify conceptually similar existing architectures and quickly interpret them to the chosen application' [8]. We intend to build on this principle to develop an initial set of patterns for architectures suitable for SoS development. The rest of this paper is laid out as follows: in Section II we discuss the requirements for an SoS architectural pattern. In Section III we summarise some relevant literature and in Section IV we introduce our approach to modelling patterns. Section V describes the initial set of patterns we have identified. Section VI outlines our initial conclusions and our future plans for extending and developing the initial collection.

This paper presents a subset of research carried out on the COMPASS[1] project into model-based techniques suitable for SoSE. Further details of architectural patterns can be found in [9].

II. Architectural principles for SoS

There are many definitions and uses of the term architecture in the relevant literature. We define architecture as 'the structure of components, their relationships, and the principles and guidelines governing their design evolution over time' (taken from IEEE Std 610.12 and DoDAF[2]).

The characteristics of SoSs described in Section I create challenges for the SoS architect, including:

- Accurately predicting and accounting for all the possible emergent behaviours is prohibitively time-consuming, or is not possible due to lack of information disclosure by CSs.
- There are very long lifecycles and the presence of legacy or COTS components that cannot be adapted to enable optimal solutions.
- CSs have other pressures to evolve outside the SoS.
- There is a high degree of technical and managerial complexity.
- There is often no central decision-making authority.

[1] http://www.compass-research.eu/
[2] http://dodcio.defense.gov/dodaf20.aspx

- SoS boundaries are blurred. The SoS tolerates the inclusion of third-party, independent CSs, and in some cases it is not clear which systems can be considered constituents and which are environment; for many SoSs this will vary depending on the current viewpoint.
- SoSs inhabit a multi-disciplinary, cross-domain world, which makes misunderstandings and functional gaps more likely.
- Socio-technical issues can complicate the system. The (unpredictable) users interacting within an SoS may be playing the role of the 'glue' enabling CSs to interoperate.
- There are commonly commercial restrictions that often impede full information disclosure between CSs.

A full collection of architectural patterns suitable for SoS should support strategies for coping with these issues.

III. Related work

In this section we introduce some relevant research in a variety of areas.

A. Architectural styles

An architectural style 'defines a family of such systems in terms of a pattern of structural organization' [10]. This facilitates reasoning and understanding about a system's design [11]. Systems can benefit from 'lessons learned' on previous systems with similar architectural properties [12], and standard architectural frameworks can be leveraged, thus increasing interoperability [13]. A range of architectural styles have been suggested for distributed systems (for example, see [14]). Many of these could be adapted for SoS engineering (SoSE). We build on the principle to develop the notion of architectural patterns for SoS in Section V.

B. Architectural description languages

An architecture description language (ADL) provides a notation or vocabulary for describing system structure and architecture [15]. A wide range of ADLs are available [12], each designed to describe different architectural concerns and/or problem domains. A single SoS is likely to see variety between the CSs, each with a different preferred ADLs. For SoS modelling, it is useful to be able to support a wide variety of concerns and/or domains. Recently there has been an interest in ADLs which support diverse semantics (i.e., which support analysis and/or code generation from descriptions in different languages). Examples include ACME [16]–[18]; xADL [16], [19] [17], [18]; and a toolset proposed by [17].

C. Architectural frameworks and component-based software engineering

Within SoSE, constituent independence, lack of central authority, long lifecycles, complexity and constant evolution create a demand for standards and/or frameworks [20] as a coping strategy. Many CSs within one SoS will already have developed their own architectural models and styles, so a framework must cope with heterogeneity. Experience from the component-based software engineering (CBSE) community can be helpful here. Although most research into CBSE so far concentrates on single systems, the CBSE field emphasises standards and frameworks to enable disparate components to communicate, and the easy substitution of one component for another [21], [22]. For example, a reference framework for Component-Based SoS is proposed by [18], supporting heterogeneous architecture, whilst [23] have proposed a domain-specific development infrastructure that supports heterogeneity.

Development and design of a component-based system is complex, and the capabilities to be supplied by legacy components or by commercial off-the-shelf (COTS) components must be taken into account alongside conventional considerations [24]. A significant source of problems in CBSE are due to 'architectural mismatches' [25], which are incorrect architectural assumptions made by CSs. This can be a problem for SoSs, where full information is often not disclosed, some CSs may be unaware of the SoS or even competing with each other, and CSs interact with legacy systems designed for outmoded architectures.

D. Enterprise architectural techniques

Enterprise architectural techniques merge systems engineering and enterprise engineering practices. Enterprise engineering has been defined as 'applying holistic thinking to conceptually design, evaluate and select a preferred structure for a future state enterprise to realise its value proposition and desired behaviours' [26]. Some key SoS challenges in enterprise architectures are: adding or removing CSs; changes in the socio-technical environment; and shifts in the enterprise profile [27]. An epoch-based analysis method has been proposed by [27], for evaluating enterprise architectures in a changing environment. A series of 'epochs' represents the system, each defining a period in time. Epoch-based architecture approaches encourage thinking about the environments of the SoS and do not focus on a single vision of the future.

E. Dynamic reconfiguration techniques

The scale and complexity of an SoS, constituent independence, constant evolution, distribution and lack of trust leads to challenging operating environments, and an SoS typically cannot be rebooted easily if there are problems. Instead, dynamic reconfiguration is one option that can be employed to cope with problems. This requires that compromised CSs can be readily substituted; a key challenge is to ensure that dynamic reconfiguration does not compromise the system architectural style (see, for example [28]). Dynamic reconfiguration is related to 'autonomic computing', which describes a system which can self-manage to some degree. Self-management approaches vary; some may emphasise collaborative working between components whilst others employ a central authority to make decisions [29]. A major challenge is the co-ordination of disparate types of system elements (e.g., databases, routers, servers...) as they monitor themselves and their neighbours, and select appropriate responses [29].

F. Design by contract

The 'design by contract' principle [30] formalises interactions between components, with one providing a service to be consumed by the other. The contract guarantees that, 'given a state and inputs which satisfy the precondition, the operation will terminate and will return a result that satisfies the post-condition and respects any required invariant properties' [15]. Contract-based engineering is particularly useful for SoSE because:

- CSs can easily be substituted, since any operation can be replaced by a similar operation as long as it has weaker or equivalent preconditions and stronger or equivalent post-conditions [15].
- CSs are able to evaluate and make decisions about the reliability of services before employing them, because details are provided about what a service will do [31].
- Expectations about services are made clear to CS developers [15].
- Contracts can facilitate dynamic reconfiguration [31].

Beugnard et al [31] apply the design-by-contract principle to architectures in which components provide services [15] in a layered approach, formally specifying synchronisation for operations and quality of service, which enables dynamic adaptation [31]. Payne & Fitzgerald [32] extend SysML into SysMLC, as a notation for specifying contract-based interfaces, that supports integration of functional and non-functional properties.

G. Summarising existing techniques

Whilst there has not been a lot of research in the area of architectural patterns for SoS explicitly, there are many areas where existing knowledge and techniques can be leveraged and possibly adapted. Research in the field of software architecture suggests that architectures are core to systems and SoS engineering. Architecting takes place throughout the system life cycle and resulting architectures must evolve over time - particularly for SoS, where change is constant. Good practice suggests that architectures be produced according to defined architectural viewpoints and codified in an architectural framework that includes consistency rules. Modelling is a key tool for supporting the development of architectures; our current and future work builds on this.

Architectural styles and patterns should be used whenever possible and architectural reuse encouraged. This increases system comprehension and allows 'lessons learned' from previous systems. SoS architectures need to support specific properties, including independence, heterogeneity, evolution and a challenging environment. There are many techniques which can be useful here. In our work on architectural patterns for SoS, we intend to support design by contract, which we believe offers an effective way to facilitate: substitution of constituents (and therefore dynamic reconfiguration as a solution to environmental problems); and loose coupling (which enables the independent constituents to evolve individually). Future work on the COMPASS project will concentrate on addressing heterogeneity in SoSE and architectural modelling.

IV. Modelling Patterns

Patterns in architecture have been studied since the 1970s (e.g., [33]). Subsequent work has led to the concept of a pattern - 'an idea that has been useful in one practical context and will probably be useful in others' [34] - being adopted in software engineering and object-oriented programming (e.g., through the work of [35]), analysis [34] and data modelling [36]. We use the term 'modelling pattern' to refer to a pattern that can be applied to modelling aspects of a system, such as its architecture or its interfaces. Following the example of [35], we describe a pattern by identifying the following basic properties: background; aims (or intent) of the pattern; pattern structure; rationale; and finally an illustrative example. We also indicate the type of SoS that we believe applies; SoS types are described in [37].

Architectural styles, design and patterns overlap significantly, but *design patterns* (e.g., see [35]) and *architectural styles* are distinct. An architectural style provides a high-level view that enables the analysis of 'emergent system wide properties' [13], whilst design patterns are concerned much more with lower-level questions. We focus on existing patterns from literature, and consider how they may be applied in a SoS.

V. Initial SoS architectural modelling patterns

We introduce in this section our initial collection of SoS architectural modelling patterns.

A. Centralised Architecture Pattern

Background A centralised architecture (similar to the star network pattern) has a central point of control. The central CS (the 'hub') is connected to the other CSs and is responsible for ensuring SoS behaviour. There may be degrees of centralisation; e.g., a fully centralised SoS will connect all CSs to a hub, whilst a partially-centralised SoS will see a hub connected to a subset of CSs. Constituents are still capable of exhibiting autonomy despite the centralisation. For example, the hub can make decisions about functionality, whilst CSs may be unaware of the SoS (e.g., they may be commercial off-the-shelf systems) and their ability to make autonomous decisions continues unabated.

Aims The main aims of this pattern are to support:

- Centralised control and management of SoS
- Reuse of pre-existing systems

Structure A single CS hub marshals the remaining CSs to deliver SoS capability. The hub is typically developed explicitly to achieve the SoS goals and so this is likely to be a directed [2] or acknowledged [37] SoS. The remaining constituents may be legacy or pre-existing systems, or purpose-built. It is typically the responsibility of the central system to ensure compatibility with the other CSs.

We distinguish between a centralised architecture with a hub, and a CS which is commonly employed by several other CSs to provide a service; the latter enacts no (or little) control and does not address the SoS goals. When designing a centralised SoS architecture, it should be made clear (e.g.,

through SysML behavioural models) that the hub provides this control.

A hub may connect to a CS which is itself considered a mini-hub in a sub-SoS, controlling/managing another collection of constituents. In a hybrid centralised-distributed SoS, the central hub may be distributed over several constituents. This provides explicit management of the SoS, but lessens the reliance placed on a single hub. The degree to which an SoS may have a distributed 'hub' and still be considered centralised is subtle; we will explore this in future work.

Rationale This pattern aims for control and management of SoS, achieved through a bespoke hub; this also permits verification in the early stages of SoS design. The pattern leverages existing or third-party systems, as the non-hub CSs can be existing or off-the-shelf systems.

Example Examples of a centralised SoS often inhabit a domain with a strong requirement for 'command and control'. The anti-guerrilla operations SoS example, described in [38], has a central 'theatre command' system (with a strong human aspect), a system comprised of UAV scouts, and CSs including artillery, troops and the required communication infrastructures. The theatre command system makes operational decisions based upon data sourced from UAV scouts and other sources, to give commands to the various troops and artillery. The goals of the SoS are achieved due to the commands of the central hub, which takes responsibility for delivering SoS functionality.

B. Service Oriented Architecture

Background Employing a Service Oriented Architecture (SOA), applications are constructed through the use of third-party services. Services are stateless from the point of view of the application (i.e., they have internal state, but do not share it), behaving like functions acting on supplied parameters. Applications are constructed by selecting services from their service description. Service providers may develop systems in any way as long as they provide standardised descriptions and expose a means for provision of the service. An SOA may be a specialisation of centralised architecture (see Section V-A).

Aims The main aims of this pattern are to support:

- Analysis of SoS emergent behaviour
- SoS/CS evolution
- Central SoS authority
- Enable cross-domain SoS development
- Long SoS lifecycle

Structure A service provider is a CS providing one or more services. Each service may be provided by one or more different providers. The SoS CSs expose interfaces, which act as points of interactions between constituents, and conform to a specified protocol, subject to a security policy. Services publish a service contract constituting a service description and a service-level agreement. The service description provides details of the service interface, in particular the offered functionality, and is used for service discovery. The service-level agreement details a set of quality of service (QoS) guarantees. A service client uses a service, through the service interface subject to the service-level agreement.

In applying SOA to SoS, we propose a similar relationship between services in SOA, and components in component-based systems (see Section III). The relationship between SOA and component-based software architectures is subtle, and often confused. In SOA, services are an abstract means to consider the functionality provided by a system; definition of the provider is the concern for software architects. Applying SOA to the SoS level, we suggest that collections of CSs may be combined to provide services[3].

When designing an SoS with the SOA pattern, we would require two separate notions of an interface: a CS interface and a service interface. The design process should first consider only services (i.e., which services are required to provide the requisite SoS functionality), not the individual CSs. An SoS designed with the SOA pattern therefore is centralized in nature, and can be considered to be directed [2] or acknowledged [37]. This is because there will be a system involved with the selection of services.

The service interfaces define the functionality provided by services. The relationship between the CS interfaces and the service interfaces is important and is the focus of further work. Each service must define a service contract. The contract should include a description of the service functionality, reflecting the service interface definition. The contract should also define a service-level agreement defining QoS guarantees. It is possible, therefore, for a service to have several service contracts for the same service interface. The contracts may differ by their service-level agreements. For example, a mapping service may have two contracts: one for civilian use and another for industry. The contracts may describe the same functionality, but vary by providing different image resolutions or response times.

Rationale As a type of centralised pattern, the SOA pattern provides a central system, which composes services to achieve the SoS goals and functionality. This satisfies the aim of central SoS authority. SoS evolution is enabled through the loose coupling of SoS management; service providers need not know how a service is being used, only that they must meet the guarantees made in the service contract. This allows the SoS to evolve through dynamic binding of services.

Through the separation of the service contract (and in particular the service interface) and the underlying service implementation, the pattern achieves the aims of CS evolution and cross-domain SoS development. A service consumer may use a service without requiring knowledge of the service logic or implementation. This enables evolution of the architecture of the CSs providing the service without requiring a change of service contract.

The analysis of SoS emergent behaviour may be achieved through the analysis of the service descriptions - both the functional and non-functional aspects. However this requires

[3]There are several areas of research in the application of SOA to SoS, including the IMC-AESOP project: http://www.imc-aesop.eu

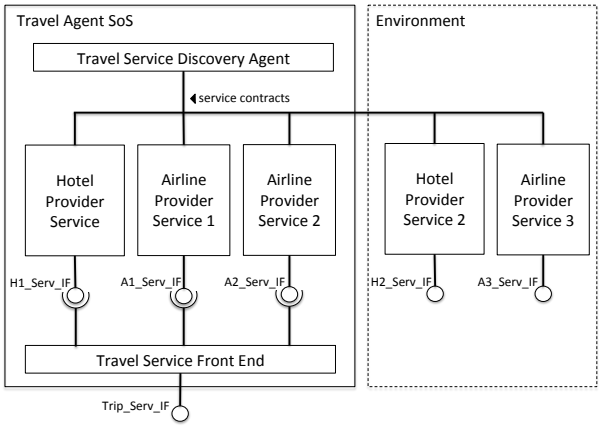

Fig. 1. Travel agency SoS implementing an SOA architectural pattern

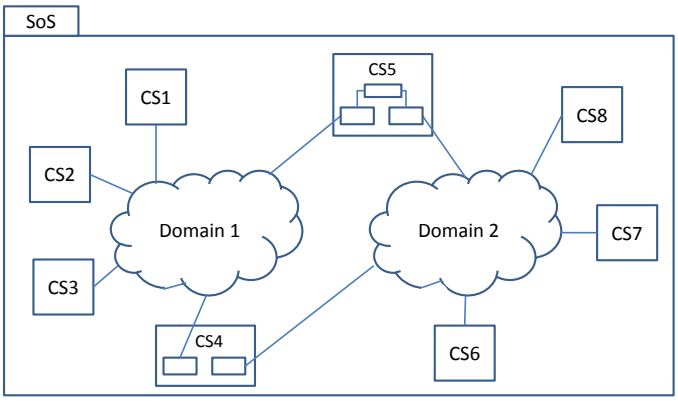

Fig. 2. Example of a publish-subscribe model with two domains

further effort in understanding the relationships between system and service interfaces, as described earlier. Finally, the SOA pattern supports a long SoS lifecycle through the use of explicit separation of service interface and service implementation, allowing CS developers to apply different methodologies and development processes.

Example A travel agent booking SoS has a central front-end system that receives requests from its environment (either a consumer or travel agent) to book a trip consisting of a hotel, flight, etc (shown in Figure 1). The front-end system is responsible for delivering the SoS functionality of receiving trip requests and responding with trip details, in the process employing those services provided by other third-party systems. A discovery system retrieves service contracts from service providers and ensures that the front end system employs the most suitable services. The front end may change service providers dynamically by selecting different suppliers to fulfil requests. The loose coupling of the SOA pattern ensures that the SoS is amenable to such reconfigurations[4].

C. Publish-Subscribe Architecture Pattern

Background This communication paradigm was developed for dissemination of information between distributed software-centric systems. The pattern can be divided in two different subgroups: an Event-Based Publish-Subscribe pattern (EBPS) and a Data-Centric Publish-Subscribe (DCPS) pattern. We introduce the event-based publish-subscribe version here, but currently focus on the data-centric publish-subscribe version.

The Publish-Subscribe pattern has been widely used in industrial systems and recently the DCPS paradigm has been standardized by OMG as the 'Data Distribution Service for Real-Time Systems' (DDS) standard [39]. DDS is a standard for a data-centric architecture for dissemination of information between heterogeneous distributed systems. DDS specifies a Data-Centric Publish-Subscribe (DCPS) model, where DCPS provides the functionality required for an application to publish and subscribe to the values of data objects of given types.

Aims The main aims of this (DCPS) pattern are to support:

- Loose coupling between publisher and subscriber CSs in the SoS with time, flow and space decoupling between the CSs
- One to many and many to many communications between the CSs in the SoS

Structure In the EBPS pattern events are exchanged between Publishers and Subscribers based on service-oriented publish and subscribe interfaces. The Publisher is responsible for publishing an event via an event channel to the associated Subscribers who have registered for the actual event type. A Subscriber receives the published events from the event channel, which handles registration of Subscribers and actual dissemination of a given event.

In the Data-Centric Publish-Subscribe (DCPS) pattern, a Topic describes a Data-Object with a unique name in the given domain, a data-type and a set of Quality of Services (QoS) related to the data. A Publisher is responsible for data distribution to a set of registered Subscribers and publishes data on one or more Topics. A DataWriter is used by the Publisher to publish data associated with a unique Topic. A Subscriber receives published data on Topics and makes these available to the application at the Subscriber site. A DataReader is used to access the received data from the attached Subscriber. Both a DataWriter and a DataReader have typed interfaces for a given topic, acting as a mediator on the publishing site to the publisher and on the receiver site to the subscriber. A Publisher, Subscriber, Topic, DataWriter or DataReader can each have an associated set of QoSPolicy. Each CS in this type of SoS can play the role of a Publisher, a Subscriber, or both, on one or more Topics. A given SoS can be defined as one or more communication domains each with its own set of CSs, as indicated on Figure 2. In this

[4]Reconfiguration challenges are still present, such as quiescence (the systems should be in a state such that transactions are not currently taking place). This is an issue beyond the scope of our current work, and not restricted to the SOA pattern.

example, constituents CS1-CS5 participate in domain 1 and constituents CS4-CS8 in domain 2. CS4 and CS5 participate in both domains; CS5 provides a potential gateway between the two domains if needed, whilst CS4 participates in the two domains without interactions between the domains. This pattern can be useful for a collaborative [2] SoS; each CS agrees on a common data model, comprising a set of topics for exchanging data. There is no central hub. If a CS Subscriber wants to join or leave a given communication domain, it can register or de-register on a given Topic in the domain.

Rationale We consider here how the solution structure addresses the aims of the DCPS pattern. Time decoupling is achieved: the Publisher and Subscribers do not need to be online at the same time (the middleware system can store data). Flow decoupling is achieved, as a publisher can publish its data asynchronously without waiting for a subscriber to receive it, whilst a subscriber does not need to wait but can be notified asynchronously when a change in subscribed data occurs. Space decoupling is achieved as neither the publisher or the subscriber needs to know each other's identity. The second aim, one-to-many and many-to-many communication, is also achieved; there can be one or more publishers on the same topic and one to many subscribers on the same topic.

Example A Medicine Card topic contains the actual medicine prescriptions for a given citizen along with the history of prescriptions. Several CSs can update this information as Publishers to the Medicine Card information whilst other CSs subscribe to the Medicine Card information and receive updates when the medicine prescription changes for a specific citizen. The underlying interaction mechanism is a push-mechanism, where changed information is pushed to all the registered subscribers. This pattern has a very loose coupling between publishers and subscribers, where it is very simple to add new publishers or subscribers.

D. Pipes and Filters Architecture Pattern

Background The pipes and filters patterns is described in [40] as an architecture style and in [41] as an architectural pattern.

Aims The main aims of this pattern are to support:

- Data or material flow oriented systems
- Independent processing steps on a flow
- Configurable transmission of the flow between processing elements
- Dynamic change of processing steps and connectors

Structure Filters represents the processing steps, where e.g. data or materials are processed from one input form to an output form. Pipes represents connections between Filters for transferring the data. The Input Source represents the first step where data enters the SoS and the Output Sink indicates where data exits. Filters are independent and do not share state or know each other's identities.

This pattern can be applied for either directed [2] or acknowledged [37] SoS, and in principle could also be used for collaborative [2] SoS if a CS voluntarily joins a processing chain to provide additional services. Each CS can act as a filter-CS (processing), a pipe-CS (storage or communication), an input-source-CS or an output-sink-CS. The complete SoS is described by the set of CS connected as: input-source-CS, {pipe-CS, filter-CS}*, pipe-CS, output-source-CS. The sets {pipe-CS, filter-CS} can be dynamically changed during the system lifetime. A modified pipes and filters system (inspired by the Modified Pipes and Filters Model [40]) is shown in Figure 3. This allows configuration of filters and/or as an alternative addition of materials to the processing steps. The SoS-Control & Configuration-CS can be used to implement either a directed or acknowledged SoS, where this CS can enforce global control over how the Filter-CS processing is performed. Another modification to the basic pattern may permit a filter to have either more than one input pipe or output pipe. This pattern modification could be combined with the control & configuration possibility, with the SoS-Control & Configuration CS deciding upon the number of input and output channels.

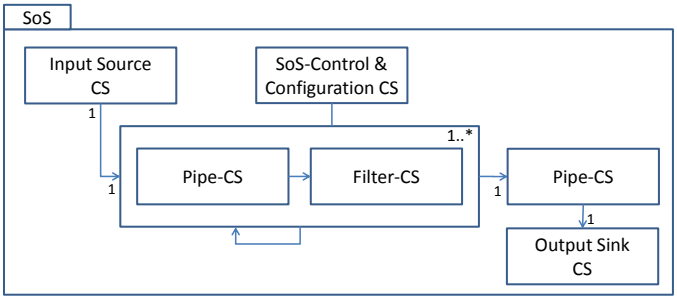

Fig. 3. Modified SoS pipe and filter pattern with control & configuration

Rationale Independent processing steps on a flow is realized as a series of independent CSs. The aim of configurable transmission of the flow is realized by the pipe CSs. The aim of supporting dynamic change of processing steps and connectors is also fulfilled, as both filters and pipes can be exchanged at design time, or sometimes dynamically during runtime.

Example A Local Monitoring system monitors a patient's vital signs in a local setting, with the possibility of storing and/or displaying results and giving local alarms. The signals monitored are forwarded through a Pipe constituent system to a Central Monitoring system, for further analysis (e.g., monitoring by a doctor). The Central Monitoring CS receives inputs from many other CSs. The processed signals can be forwarded to another CS performing the role of a Pipe for Central Storage. In some situations Pipe CSs could be implemented by standard middleware, thus disappearing as independent CSs. This architecture ensures that the CS are decoupled, as the receivers of a given flow can be redirected dynamically to another Filter component.

E. Blackboard Architecture Pattern

Background The Blackboard architecture pattern is described by [40] as an architectural style for data-centred

Systems and in [41] as an architectural pattern. It is suitable for SoSs solving problems with a certain degree of uncertainty, e.g. in expert based or fuzzy logic based systems. The pattern can also be applied in SoS where central knowledge is obtained by several sources.

Aims The main aims of this pattern are to support:

- Development of expert or knowledge based systems
- Loose coupling
- Separation of concerns

Structure The components are the Knowledge Sources, a Blackboard data structure and a Control component. The Blackboard component is a central data store, where elements of the solution space are stored together with control data. The Blackboard provides an interface for reading and writing data. Elements of the solution space are written to (or removed from) the Blackboard by the Knowledge Source components; the elements are called Hypotheses. Knowledge Source CSs are specialized for either solving a part of the overall problem or delivering input data; they work independently and usually in parallel. Each has a condition part, that evaluates the Blackboard state to see if it can make a contribution, and an action part, that produces a result and updates the blackboard state.

The Control component evaluates the current state of the blackboard and uses this data to coordinate the Knowledge Sources. The Control component searches for a possible solution to the problem, which cannot always be guaranteed. This pattern is applicable for directed [2], acknowledged [37] and possibly also for collaborative [2] SoSs. Each CS acts as a Knowledge Source, generating information which is stored in the Blackboard CS. Application of this pattern requires one or two CSs acting as the repository (Blackboard) and control system, with a specification of the interaction between them.

Rationale The Blackboard component houses the 'expert' constituent system, where the information and hypotheses are stored and modified by the Knowledge Source constituent systems. Loose coupling is achieved, through the independence of the Knowledge Sources, which may be unaware of the existence of other sources (the SoS-Control constituent is coupled with the Blackboard and Knowledge Source constituents). The aim of separation of concerns is achieved through the use of three different system roles. A Knowledge Source is free to pursue its own goals outside the SoS, including in other SoSs.

Example RadarSat-1 (based on a real-life example called RadarSat-1 [42]) is an earth-observation satellite (shown in Figure 4), equipped with an aperture radar to allow end users to connect, submit requests and receive the results. A blackboard is used to realize an advanced planning component, which controls the radar measurements. This system has over 140 constraints, which makes the planning process very complex. The SoS-RadarSat-CS is a CS and controls the access and use of the satellite radar, which is a shared resource in the community of RadarSat users. Each Client-CS in this SoS is an independent system with its own purpose that participates in the SoS as a Knowledge Source. RadarSat SoS allows each

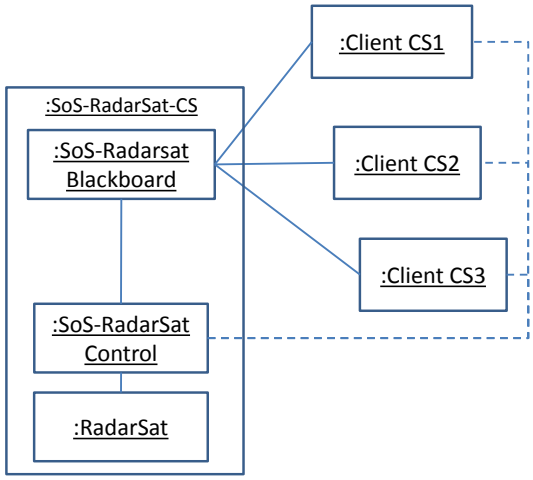

Fig. 4. A satellite SoS implementing a blackboard pattern

Client CS to perform its own measurements and experiments with the possibility of sharing the results in the community.

VI. Conclusions and Future Work

In this paper we have presented an initial set of patterns for architectures suitable for SoSs. We suggest that better understanding of SoS patterns supports architectural reuse, increasing system comprehension and leverage of 'lessons learned' for SoS engineers.

A key feature for SoS architectural modelling is causal or timed sequences. These are important for SoS modelling, to illustrate how:

- The architectural pattern of an SoS may evolve over a long period of time (e.g., CSs added to or removed from the SoS, or changes to dependencies between CSs).
- The SoS responds to events by reconfiguring the architecture dynamically, taking advantage of new services, or replacing unavailable CSs. This includes consideration of:
  - New emergent properties at the SoS boundary
  - Dynamic contract negotiations. During this process third-party agents acting on behalf of CSs may make dynamic changes to the SoS contracts, relating to CSs' functionality or the related non-functional properties.
  - Control structures required for enacting system changes (either behavioural or structural) based on the state of the SoS and its environment.

In future work we intend to consider causal and timed sequences in architectural modelling for SoSs, taking into account the issues described above, as well considering methods for coping with heterogeneity and/or methods for identifying the events that trigger a transition between different architectural patterns. Published case studies that implement recognisable SoS patterns are needed, so our planned future work also includes analysis of SoS case studies, expanding our initial collection with additional patterns where appropriate. Finally, based on case study analysis, we should also like to

develop practical guidance for use of possible patterns based on the characteristics of specific SoSs.

ACKNOWLEDGMENT

The work presented here is supported by the EU Framework 7 Integrated Project 'Comprehensive Modelling for Advanced Systems of Systems' (COMPASS, Grant Agreement 287829). For more information see http://www.compass-research.eu.

REFERENCES

[1] OUSD(AT&L), DoD, "Systems and Software Engineering. Systems Engineering Guide for Systems of Systems," Office of the Deputy Under Secretary of Defense for Acquisition and Technology, Department of Defense, Washington DC, Tech. Rep. Version 1.0., August 2008.

[2] M. W. Maier, "Architecting Principles for Systems-of-Systems," *Systems Engineering*, vol. 1, no. 4, pp. 267–284, 1998.

[3] D. A. Fisher, "An Emergent Perspective on Interoperation in Systems of Systems," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep., March 2006, cMU/SEI-2006-TR-003.

[4] J. Boardman and B. Sauser, "System of Systems – the meaning of "of"," in *Proceedings of the 2006 IEEE/SMC International Conference on System of Systems Engineering*. Los Angeles, CA: IEEE, April 2006, pp. 118–123.

[5] W. Baldwin and B. Sauser, "Modeling the Characteristics of System of Systems," in *System of Systems Engineering, 2009. SoSE 2009. IEEE International Conference on*. IEEE, 2009, pp. 1–6.

[6] R. Abbott, "Open at the Top; Open at the Bottom; and Continually (but slowly) Evolving," in *System of Systems Engineering, 2006 IEEE/SMC International Conference on*. IEEE, April 2006.

[7] D. Cocks, "How Should We Use the Term "System of Systems" and Why Should We Care?" in *Proceedings of the 16th INCOSE International Symposium 2006*. INCOSE, July 2006.

[8] C. Dickerson and D. N. Mavris, *Architecture and Principles of Systems Engineering*. CRC Press, 2009.

[9] S. Perry, J. Holt, R. Payne, C. Ingram, A. Miyazawa, F. O. Hansen, L. D. Couto, S. Hallersteded, A. K. Malmos, J. Iyoda, M. Cornelio, and J. Peleska, "Report on modelling patterns for sos architectures," COMPASS Deliverable, D22.3, Tech. Rep. [Online]. Available: http://www.compass-research.eu/deliverables.html

[10] D. Garlan and M. Shaw, "An introduction to software architecture. Technical report: CMU/SEI-94-TR-21," Software Engineering Institute, Carnegie Mellon University, Tech. Rep., 1994.

[11] D. Garlan, "Software architecture: a roadmap," in *In Proceedings of the Conference on the Future of Software Engineering: ICSE00*, 2000, pp. pp91–101.

[12] M. Shaw, "The coming-of-age of software architecture research," in *In Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, 2001, pp. 656–664a.

[13] R. Monroe, A. Kompanek, R. Metlon, and D. Garlan, "Architectural styles, design patterns, and objects," *IEEE Software*, vol. 14, no. 1, pp. 43–52, 1997.

[14] C. Weir, "Architectural styles for distribution: Using macro-patterns for system design," in *In Proceedings of the 1997 European Pattern Languages of Programming Conference, Irsee. Available as Siemens Technical Report 120/SW1/FB*, 1997.

[15] R. J. Payne and J. S. Fitzgerald, "Evaluation of Architectural Frameworks Supporting Contract-based Specification," School of Computing Science, Newcastle University, Tech. Rep. CS-TR-1233, December 2010.

[16] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "A comprehensive approach for the development of modular software architecture description languages," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 2, 2005.

[17] M. Leclercq, A. E. Ozcan, V. Quema, and J. Stefani, "Supporting heterogeneous architecture descriptions in an extensible toolset," in *In Proceedings of the International Conference on Software Engineering: ICSE*, 2007, pp. 209–219.

[18] L. S. Frederic Loiret, Romain Rouvoy and P. Merle, "Software Engineering of Component-based System-of-Systems: A Reference Framework," in *CBSE '11 14th international ACM Sigsoft symposium on Component based software engineering*. ACM, 2011, pp. 61–65.

[19] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "An infrastructure for the rapid development of XML-based architecture description languages," in *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, 2002.

[20] S. A. Selberg and M. A. Austin, "Toward an evolutionary system of systems architecture," in *In Proceedings of the 18th Annual International Symposium of The International Council on Systems Engineering (INCOSE)*, June 2008.

[21] W. Kozaczynski and G. Booch, "Component-based software engineering," *IEEE Software*, vol. 15, no. 5, pp. 34–26, 1998.

[22] H. Jifeng, X. Li, and Z. Liu, "Component-based software engineering: The need to link methods and their theories," in *ICTAC 2005*, ser. LNCS 3722, D. V. Hung and M. Wirsing, Eds., 2005, pp. 70–95.

[23] G. Edwards and N. Medvidovic, "A methodology and framework for creating domain-specific development infrastructures," in *In Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering: ASE*, 2008, pp. 168–177.

[24] W. Hasselbring, *Component-based software engineering*. World Scientific Publishing, 2002, vol. 2, pp. 289–305.

[25] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch or why its hard to build systems out of existing parts," in *Proceedings of the 17th International Conference of Software Engineering (ICSE 95)*, 1995.

[26] D. Nightingale and D. Rhodes, "Enterprise architecting: Course notes. MIT ESD-38J," Massachussetts Institute of Technology, Tech. Rep., 2007.

[27] D. Rhodes, A. Ross, and D. Nightingale, "Architecting the system of systems enterprise: Enabling constructs and methods from the field of engineering systems," in *Systems Conference, 2009 3rd Annual IEEE*, march 2009, pp. 190 –195.

[28] I. Georgiadis, J. Magee, and J. Kramer, "Self-organising software architectures for distributed systems," in *In Proceedings of the first workshop on Self-healing systems (WOSS 02)*, 2002, pp. 33–28.

[29] J. O. Kephart, "Research challenges of autonomic computing," in *In Proceedings of the 27th International Conference on Software Engineering (ICSE)*, 2005, pp. 15–22.

[30] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[31] A. Beugnard, J.-M. Jezequel, N. Plouzeau, and D. Watkins, "Making Components Contract Aware," *IEEE Computer*, vol. 32, no. 7, pp. 38–45, July 1999.

[32] R. J. Payne and J. S. Fitzgerald, "Contract-based interface specification language for functional and non-functional properties," School of Computing Science, Newcastle University, Tech. Rep. CS-TR-1250, May 2011.

[33] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 1977.

[34] M. Fowler, *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.

[36] D. Hay, *Data Model Patterns: Conventions of Thought*. Dorset House, 1996.

[37] J. Dahmann and K. Baldwin, "Understanding the Current State of US Defense Systems of Systems and the Implications for Systems Engineering," in *IEEE Systems Conference*. IEEE, April 2008.

[38] M. Hall-May and T. P. Kelly, "Using agent-based modelling approaches to support the development of safety policy for systems of systems," in *Proceedings of the 25th International Conference on Computer Safety, Reliability and Security (SAFECOMP 06)*, ser. LNCS, J. Gorski, Ed., vol. 4166, Sep 2006, pp. 330–343.

[39] OMG, "Data distribution service for real time systems, specification, version 1.2," http://www.omg.org/spec/DDS/1.2 (Accessed November 2012), January 2007.

[40] M. Shaw and G. Garlan, *Software Architecture Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[41] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture a system of patterns*. John Wiley & Sons, 1996, vol. 1.

[42] D. D. Corkill, "Countdown to success: Dynamic objects, GBB, and RADARSAT-1," *Communication of the ACM*, vol. 40, no. 5, May 1997.