# Services, SOAs and Integration at Scale[1]

Ingolf Krüger
University of California, San Diego
La Jolla, CA 92093-0404
USA

**Abstract** After more than a decade of research into and practical application of service-orientation (SO*) there is still confusion about what services are, and what their benefit to Software Engineering research and practice is. Nevertheless, and unfazed by its fuzzy backdrop, SO* has taken hold across industry as the foundation on which large software systems are built. Clearly, this leaves many fundamental questions on the table.

This paper focuses on developing the principles and practices for using SO* in large-scale service integration. We start from a simple premise: *services are functions, service-oriented architectures (SOAs) are dynamic functional programs*. We develop this premise into a semantic foundation for services and SOAs based on a novel dynamic model of stream processing functions. We introduce Open Rich Services (ORS), an architecture pattern for SOA that (a) disentangles infrastructure-concerns cleanly from application-specific concerns, (b) supports flexible and dynamically changeable service composition, and (c) facilitates hierarchical service decomposition. We establish the link between ORS and our basic SOA semantics to yield a comprehensive and scalable SOA foundation. As a proof of concept we develop a domain specific language (DSL) for specification of SOAs following this semantic approach.

[1] This paper was written to a significant extent while on sabbatical from UCSD, visiting Technical University of Munich, Saarbrücken University, Technical University of Aachen, and Paderborn University. The author is grateful to his hosts (Profs. Broy, Finkbeiner, Rumpe and Schäfer, respectively) and their institutions for providing a highly stimulating and productive research environment.

## Table of Contents


1. Introduction ..... 3
   1.1. Existing Definitions for Services and SOAs ..... 3
   1.2. The OASIS Reference Model for Service Oriented Architecture ..... 4
   1.3. Approach ..... 6
   1.4. Contributions and Outline ..... 7
2. Streams and Stream Processing Functions ..... 8
   2.1. Notational conventions ..... 8
   2.2. System Structure ..... 10
   2.3. System Behavior ..... 10
   2.4. Stream Processing Functions ..... 11
   2.5. Refinement of Stream Processing Functions ..... 12
3. Semantics of Services and SOAs ..... 13
   3.1. Services ..... 13
   3.2. Components ..... 13
   3.3. Service-Oriented Architectures (SOAs) ..... 13
   3.4. Observations and Discussion ..... 15
4. Rich Services: SOAs at Scale ..... 16
   4.1. Rich Service Semantics ..... 17
   4.2. Semantics of Hierarchical Rich Services ..... 18
   4.3. Observations and Discussion ..... 19
5. Open Rich Services (ORS): A DSL for SOA at Scale ..... 20
   5.1. Example: CellSense ..... 21
   5.2. What makes CellSense Challenging? ..... 22
   5.3. Open Rich Services DSL in Clojure ..... 24
   5.4. The ORS Runtime System ..... 33
   5.5. Observations and Discussion ..... 34
6. Discussion and Related Work ..... 35
7. Summary & Outlook ..... 39
8. Acknowledgments ..... 39
9. References ..... 40

# 1. Introduction

Service-Orientation (SO*)[2] has become the de-facto industry standard for the development of Internet-scale systems. Its main value proposition is reduced cost and risk while facilitating rapid, flexible and dynamic construction, composition, deployment, quality assurance and maintenance of individual, finely-granular features into a multitude of systems. The resulting systems typically are "open" in the sense that they can become services themselves allowing further composition with yet other services.

While these qualities have always been important, today's fast innovation cycles with rapid requirements changes and the need to equally rapidly respond with changes in existing, or the development of new products, have further amplified their significance. This has catapulted SO* into being the "go-to" approach for software and systems integration.

At the same time, no clear consensus on what services and service-oriented architectures *are* has so far emerged either in industry or academia. As a consequence, service-orientation is regarded as a fuzzy field at best, and a hype-ridden curiosity at worst. This endangers the long-term viability of SO* as a field within software and systems engineering, despite the significant promise it holds.

In this paper, we develop a concise definition of services and service-oriented architectures (SOAs) that can serve as the basis for the use of these terms *across* stakeholder groups and their projects.

## 1.1. Existing Definitions for Services and SOAs

Part of the challenge in arriving at a precise definition for services and SOAs is that the abundance of standards and concomitant opinions on what SO* should be or do is legion, each flavored by the needs and concerns of a specific stakeholder group.

We give three illustrative examples of such definitions:

1. *"SOA is an application framework that takes everyday business applications and breaks them down into individual business functions and processes, called services. SOA is a conceptual description of the structure of a software system in terms of its components and the services they provide, without regard for the*

[2] We use the abbreviation SO* to refer to the conglomerate of terms (or a subset thereof) that has emerged around service-orientation: services themselves, service-oriented architectures, design, development, deployment, … From the context, the reader will be able to deduce what concrete set of terms a particular reference to SO* intends to capture.

*underlying implementation of these components, services and connections between components." [1]*

2. *"Service: A service is an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of provider entities and requesters entities. To be used, a service must be realized by a concrete provider agent. SOA: A set of components which can be invoked, and whose interface descriptions can be published and discovered." [2]*
3. *"SOA: A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations." [3]*

We note that definition 1) comes from an industry-leading provider of consulting on and products for service-oriented systems, whereas 2) and 3) are taken from publications of standardization bodies for SO* technologies. There is nothing inherently wrong with any one of the definitions, taken in isolation. It is striking, however, how little agreement there is across definitions on the fundamental notions defining the field of SO*.

With respect to services, definitions 1)-3) say they are "individual business functions and processes", "an abstract resource that represents a capability", and "a distributed capability", respectively.

None of the definitions agree on what a SOA is. Definition 1) states that SOA is *both* "an application framework" and "a conceptual description". Definition 2) says SOA is a set of components, whereas definition 3) says that SOA is a paradigm. All three definitions complect the terms service and SOA with properties that even at second sight seem independent of whether the chosen technology or methodology is service-oriented or not. Definition 1) juxtaposes services with business functions, but surely component-orientation, structured design and object-orientation each make the same claim for their units of granularity and composition (components, modules, and objects, respectively). Similarly, definition 2) links services with "coherent functionality" between "provider entities and requestor entities", which also is a claim that is not unique to services – and thus not a distinguishing characteristic we can use to better understand what SO* is *and* is *not*. Finally, definition 3) requests that SOAs support production of "desired effects consistent with measurable preconditions and expectations", which again is by no means a desire germane to SO*.

### 1.2. The OASIS Reference Model for Service Oriented Architecture

Although clear, concise definitions of the terms service and SOA seem elusive, we need to ground the discussion of our own approach with respect to common interpretations of these terms. To that end, we look to the OASIS Reference Model for SOA [3], which identifies core concepts of and terminology surrounding SOAs:

| Concept/Term | Informal Meaning |
|---|---|
| Capability | The ability of an entity to perform an action |
| Needs | A necessity for an entity to perform its own actions. |
| Service | "Mechanism by which needs and capabilities are brought together" (ibid., lines 173-174.) |
| Visibility | "the capacity for those with needs and those with capabilities to see each other" (ibid., lines 140-141) |
| Semantics | Interpretation of data and actions within a domain of discourse. |
| Execution Context | "the set of technical and business elements that form a path between those with needs and those with capabilities" (ibid., lines 148-150) |
| Real World Effects | Observable state change within the environment of the SOA. |
| Service Provider | Entity offering a capability. |
| Service Consumer | Entity perusing a service provider to fulfill a need. |
| Service Description | "information necessary to interact with the service … in such terms as the service inputs, outputs and associated semantics." (ibid., lines 188-189) |
| Interaction | "activity of using a capability" (ibid., line 146) |
| Contract | "an agreement by two or more parties" (ibid., lines 658-659) |
| Policy | "represents some constraint or condition on the use, deployment or description of an owned entity as defined by any participant" (ibid., lines 657-658) |

Furthermore, [3] requests "any design for a system that adopts the SOA approach will

- Have entities that can be identified as services as defined by this Reference Model;
- Be able to identify how visibility is established between service providers and consumers;
- Be able to identify how interaction is mediated;

- Be able to identify how the effect of using services is understood;
- Have descriptions associated with services;
- Be able to identify the execution context required to support interaction; and
- It will be possible to identify how policies are handled and how contracts may be modeled and enforced." (ibid., lines 765-775)

Although this list does not explicitly mention the terms "scale" and "composition", [3] recognizes their significance by stating "The value of SOA is that it provides a simple scalable paradigm for organizing large networks of systems that require interoperability to realize the value inherent in the individual components." (ibid., lines 284-285)

The approach we present in the remainder of this document explicitly answers these key characteristics of SOA systems *and* is precise in their definition.

## 1.3. Approach

We seek to establish concise definitions for both services and SOAs. Our goal is to disentangle the notion of service from ancillary concerns such as relationships to business terminology, specific implementation details, or deployment concerns, while addressing the main SOA characteristics identified in Section 1.2.

We realize, of course, that some benefits of SO* arise form the interdisciplinary appeal services have, say, between software developers and managers seeking flexible code-level system integration, and more flexible product offerings with shorter time-to-market, respectively. However, contrary to the definitions quoted in Section 1.1, we advocate working from a simple characterization of services and SOAs, so that ancillary concerns can be layered on top rather than being part of the basic definitions.

What, then, are the characteristics germane to a basic yet viable service and SOA definition?

### 1.3.1. Behavior, Composition, Encapsulation, Scale

Services are units of behavior and composition in SOAs. Furthermore, for systems of realistic size, we must have ways to encapsulate services so as to support hierarchical decomposition; one proven mechanism for accomplishing this is to supply services with explicit interfaces.

Therefore, key elements of the service notion are

(1) Services themselves as units of behavior,
(2) Service interfaces to support encapsulation,
(3) Service composition to express the interplay among services,
(4) Service decomposition to support hierarchical refinement and abstraction.

Our model needs to be flexible enough such that we can distinguish closed from open systems. A closed system offers no services to its environment, whereas an open system does.

### 1.3.2. Dynamic Binding

One feature all service and SOA definitions agree on is dynamic discovery and binding of service implementations given a form of service specification or description. This facilitates loose coupling between service providers and consumers, and is thus responsible for many of the desires of development- and runtime flexibility and agility stakeholders project on SO*.

### 1.3.3. A Minimalist Service Model

Based on these observations, we work with the following intuitive understanding of what services and SOAs are:

*Services are functions, and SOAs are dynamic functional programs.*

Functions are an immediate choice for the modeling of service behavior; they come with clearly specified interfaces, function composition is well-understood, as is function decomposition (as the dual of function composition).

Functional programs, in their pure form, express computation as the composition of a set of functions. Dynamic functional programs allow the binding between function identifiers, and the functions they refer to, to change over time.

We conjecture that this minimalist characterization, which we will substantiate via a correspondingly modest set of formal definitions, captures the essence of SO*; because it is so concise it can easily be tailored for specific application domains, or general project needs.

As discussed above, we do not conflate the service notion with a particular deployment technology. Consequently, we view Web Services as just one of a family of standards and technologies capable of implementing a SOA. (cf. also [3], lines 205-209)

## 1.4. Contributions and Outline

The major contributions of this paper are:

1. A formal semantics for services and dynamic functional programs, giving rise to a formal semantics for SOAs.
2. Introduction of Open Rich Services (ORS) as a specific architectural pattern together with its formal semantics, for hierarchically scalable SOAs disentangling application- from infrastructure concerns.
3. Presentation of a domain specific language (DSL) as a prototyping environment for ORS – with a specific mapping of services and SOAs to functions and dynamic functional programs in the programming language Clojure.

In Section 2 we introduce the semantic model of streams and stream-processing functions, before we use this model to precisely characterize services and SOAs in Section 3. In Section 4, we introduce Rich Services as an architectural pattern, and base its semantics on the definitions from Section 3. In Section 5 we give a proof of concept for ORS via an embedded DSL implemented in Clojure. In Section 6 we discuss the presented semantic framework in the context of related work. We provide a summary and outlook in Section 7.

## 2. Streams and Stream Processing Functions

We now establish a few conventions and definitions in preparation of the definitions we give for services and SOAs in Section 3.

We closely follow the model of streams and stream processing functions as introduced in [4], [5] and [6].

A key benefit of this model is its conciseness while being expressive for a broad range of systems, including the distributed, reactive systems we are interested in here.

### 2.1. Notational conventions

We start with a few notational conventions. By $\mathbb{B}$ and $\mathbb{N}$ we denote the set of Booleans (the constants are true and false) and natural numbers (including 0), respectively. We define $\mathbb{N}_\infty \triangleq \mathbb{N} \cup \{\infty\}$ for the set of naturals together with their supremum $\infty$. We use the usual extensions of (binary) operations from $\mathbb{N}$ to $\mathbb{N}_\infty$; examples are $x \leq \infty$ for all $x \in \mathbb{N}_\infty$, $\max(\infty, x) = \max(x, \infty) = \infty$ and $\min(\infty, x) = \min(x, \infty) = x$ for all $x \in \mathbb{N}_\infty$. To denote function application we often use an infix dot (".") instead of parentheses to increase readability of our formulae. For $Q \in \{\forall, \exists\}$ and predicates $r$ and $p$ we write $\langle Qx : r.x : p.x \rangle$ to denote the respective quantification over all $p.x$ for which $x$ satisfies the quantification range $r.x$. If the range is understood from the context, we omit it from the quantifying formula. As another form of reduced notation we integrate simple ranges into the specification of the quantified variable; as an example, we sometimes write $\langle \forall x \in \mathbb{N} :: \ldots \rangle$ instead of $\langle \forall x : x \in \mathbb{N} : \ldots \rangle$. $\wp(X)$ denotes the powerset of any set $X$. Given sets $Y_1, Y_2, \ldots$ we define for tuples $y = (y_1, y_2, \ldots) \in Y_1 \times Y_2 \times \ldots$ the projection onto the $i$-th element of the tuple as $\pi_i.y \triangleq y_i$ for $i \geq 1$. For the closed interval between $m \in \mathbb{N}_\infty$ and $n \in \mathbb{N}_\infty$ we write $[m, n]$; if $m > n$ then $[m, n] \triangleq \varnothing$.

The mathematical model serving below as the basis for our notion of system behavior is that of streams. Streams and predicates or functions on streams are an extremely powerful specification mechanism for distributed, interactive systems (cf. [7] [8] [5] [9] [10]). It serves particularly well for property-oriented behavior specifications, as well as for the definition of refinement notions and for the verification of corresponding refinement relationships between specifications.

Here, we give a concise overview of the major concepts and notations with respect to streams to the extent required for this document; for a thorough introduction to the topic, we refer the reader to [5] and [9].

A stream is a finite or infinite sequence of messages. By $X^*$ and $X^\infty$ we denote the set of finite and infinite sequences over set $X$, respectively. $X^\omega \triangleq X^* \cup X^\infty$ denotes the set of streams over set $X$. Note that we may identify $X^*$ and $X^\infty$

with $\bigcup_{i\in\mathbb{N}}([0,i]\to X)$ and $\mathbb{N}\to X$, respectively. This allows us, for $x\in\mathbb{N}$ and $n\in\mathbb{N}$, to use function application to write $x.n$ for the $n$-th element of stream $x$. By $|x|$ we denote the length of stream $x$. It is equal to some natural number if $x\in X^*$; for $x\in X^\infty$, $|x|$ yields $\infty$. Furthermore, for $x\in X^\omega$ and $n\in\mathbb{N}$, with $|x|\geq n$, we define $x{\downarrow}n$ to be the prefix of $x$ with length $n$. By $x{\uparrow}n$ we denote the stream obtained from $x$ by removing the first $n$ elements. $x{\uparrow}\infty$ yields the empty stream. We write the concatenation of two streams $x,x'\in X^\omega$ as $x\cdot x'$. If $|x|=\infty$, then $x\cdot x'$ equals $x$. By $\langle x_1,x_2,\ldots,x_n\rangle$ we denote the finite stream consisting, in this order, of the elements $x_1$ through $x_n$ with $x_i\in X$ for $1\leq i\leq n$. For $x=\langle x_1,x_2,\ldots,x_n\rangle$ and $y\in X$ we define $y\in x\triangleq\langle\exists i:1\leq i\leq n:x_i=y\rangle$. As a shorthand, we define $x_1\cdot x\triangleq\langle x_1\rangle\cdot x$ for $x_1\in X$ and $x\in X^\omega$. We denote the empty stream by $\langle\rangle$. For an element $x_1\in X$ and $n\in\mathbb{N}_\infty$ we define $x_1^n$ to be the stream over $X$ consisting of $n$ consecutive copies of $x_1$. Given a subset $Y\subseteq X$ the projection $x|_Y$ yields the stream obtained from $x\in X^\omega$ by removing all elements not contained in $Y$. The restriction function $x|_{[m,n]}\triangleq(x{\downarrow}n){\uparrow}m$ yields the part of stream $x$ that starts at position $m$ and ends at position $n$.

We lift the operators introduced above to finite and infinite tuples and sets of streams by interpreting them in a pointwise and elementwise fashion, respectively. Given, for instance, the stream tuple $x:[1,m]\to X^\omega$, with $x=(x_1,\ldots,x_m)$ for $m\in\mathbb{N}$, we denote by $x.n$ the tuple $(x_1.n,\ldots,x_m.n)$, if $n\in\mathbb{N}$.

Below, we use streams to model the behavior of systems over time. We emphasize that messages can assume any data type; therefore, we can model both streams of messages and streams of states, depending on what behavioral concept we want to highlight. To stress this intuition, we introduce the name *timed streams* for infinite streams (time does not halt) whose elements at position $t\in\mathbb{N}$ represent the messages transmitted, or the states assumed, at time $t$. Based on this intuition, we identify tuples over timed streams with streams over tuples and call both *timed stream tuples*. For instance, we identify $(X\times Y)^\infty$ with $X^\infty\times Y^\infty$ for sets $X$ and $Y$. Moreover, for finite index sets $X$, and arbitrary sets $Y$ we identify elements of the domains $X\to Y^\infty$ and $(X\to Y)^\infty$. This technical convention gives us a convenient way of converting streams of functions into functions, whose ranges are streams and vice versa. If, as an example, we have $z\in(X\to Y)^\infty$, and $x\in X$, then we allow ourselves to write $z.x$ to obtain $z$'s projection onto $x$. Similarly, if we have $z\in(X\to Y^\infty)$ and $t\in\mathbb{N}$, then we consider $z.x.t$ and $z.t.x$ to be synonyms.

### 2.2. System Structure

Structurally, a *system* consists of a set of function identifiers $FID$ and a set $C$ of names of directed channels. Functions model processes of the system, channels model the communication histories between functions. The messages exchanged via channels come from set $M$ of messages.

Channels can be bound to function identifiers. Specifically, we assume two functions $src, dst : C \to FID$ such that $src.c = f$ or $dst.c = f$ if $f$ is the source or destination, respectively, of channel $c \in C$.

### 2.3. System Behavior

We assume that the functions comprising the system communicate between one another and their environment by exchanging messages over channels. We assume further that a discrete global clock drives the system. We model this clock by the set $\mathbb{N}$. Intuitively, at time $t \in \mathbb{N}$ every function in the system determines its output based on the messages it has received until time $t-1$. It then writes its output to the corresponding output channels. The delay of at least one time unit models the processing time between an input and the output it triggers; more precisely, the delay establishes a strict causality between an output and its triggering input (cf. [7] [6]).

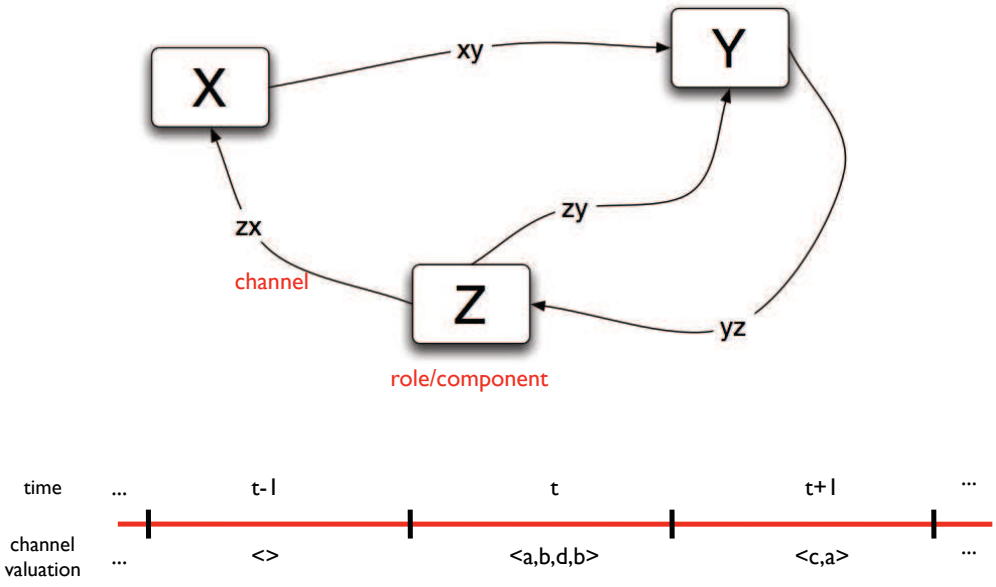


Figure 1: System structure and channel valuations.

Formally, with every channel $c \in C$ we associate the histories obtained from collecting all messages sent along $c$ in the order of their occurrence. Our basic assumption here is that communication happens asynchronously: the sender of a message does not have to wait for that message's receipt by the recipient. This allows us to model channel histories by means of streams.

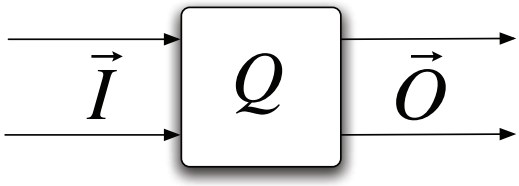


Figure 2: Syntactic interface and stream processing function

#### 2.3.1. Valuations/Histories

For a set $X \subseteq C$ and a set $M$ of messages, we define $\tilde{X} \triangleq \left(X \to M^*\right)$ as the set of valuations of the channels in $X$. By $\vec{X} \triangleq \tilde{X}^\infty$ we denote the set of *infinite valuations* or *histories* of the channels in $X$.

#### 2.3.2. Syntactic Interface

For $f \in FID$, $I_f \subseteq C$ and $O_f \subseteq C$ we define $f$'s *syntactic interface* as the pair $\left(I_f, O_f\right) \in \wp(C) \times \wp(C)$ iff

$$I_f = \{c \in C : dst.ch = f\}$$
$$O_f = \{c \in C : src.ch = f\}$$

A *directed syntactic interface* $\left(I_f, O_f\right)$ satisfies $I_f \cap O_f = \varnothing$. We use the shorthand $I_f \triangleright O_f$ for $f$'s directed syntactic interface. We omit the subscript $f$ if the function name is clear from the context.

### 2.4. Stream Processing Functions

By $\vec{I} \to \wp\left(\vec{O}\right)$ we denote the set of all set-valued functions that relate a set of *output histories* with a given *input history*. We call elements of $\vec{I} \to \wp\left(\vec{O}\right)$ *stream processing* functions over the syntactic interface $(I,O)$. Set-valued functions are relations, hence we can identify $\vec{I} \to \wp\left(\vec{O}\right)$ with $\vec{I} \to \left(\vec{O} \to \mathbb{B}\right)$, i.e. with the relation's defining predicate. We exploit this equivalence by selecting the most appropriate form depending on the style of mathematical formula we write.

#### 2.4.1. Causality

We call $Q : \vec{I} \to \wp\left(\vec{O}\right)$ *causal* if

$$\left\langle \forall t \in \mathbb{N}, x, y \in \vec{I} :: x{\downarrow}t = y{\downarrow}t \Rightarrow (F.x){\downarrow}(t+1) = (F.y){\downarrow}(t+1) \right\rangle$$

holds. The output at time $t \in \mathbb{N}$ of causal functions depends at most on the input history seen until strictly before $t$.

### 2.4.2. Domain

The domain $dom.Q$ of function $Q:\vec{I}\to\left(\vec{O}\to\mathbb{B}\right)$ is defined as

$$dom.Q \triangleq \left\{x\in\vec{I}:\left\langle\exists y\in\vec{O}::Q.x.y\right\rangle\right\}$$

A function $Q:\vec{I}\to\left(\vec{O}\to\mathbb{B}\right)$ is left-total, iff $dom.Q=\vec{I}$.

### 2.4.3. Composition

Let $F:\vec{I}_F\to(\vec{O}_F\to\mathbb{B})$ and $G:\vec{I}_G\to(\vec{O}_G\to\mathbb{B})$ be two stream processing functions with $I_F\cup O_F\cup I_G\cup O_G\subseteq C$ and $O_F\cap O_G=\varnothing$. We denote the composition of $F$ and $G$ by $F\otimes G$, and define its syntactic interface $(I,O)$ as

$$I\triangleq\left(I_F\cup I_G\right)\setminus\left(O_F\cup O_G\right)$$
$$O\triangleq\left(O_F\cup O_G\right)\setminus\left(I_F\cup I_G\right)$$

with

$$(F\otimes G).x.\left(y|_O\right)\equiv y|_I = x\wedge F.\left(y|_{I_F}\right).\left(y|_{O_F}\right)\wedge G.\left(y|_{I_G}\right).\left(y|_{O_G}\right)$$

We introduce the sequential composition of $F$ and $G$, written $F;G$, as a shorthand for $F\otimes G$ where $O_F=I_G$ and $I_F\cap O_G=\varnothing$.

## 2.5. Refinement of Stream Processing Functions

We introduce refinement notions for stream processing functions in preparation of (a) defining "compatibility" between services in a composition of multiple services, and (b) defining service hierarchies.

### 2.5.1. Property Refinement

Let $F,G:\vec{I}\to\left(\vec{O}\to\mathbb{B}\right)$ be stream processing functions. We call $G$ a property refinement of $F$, and write $G\leq_p F$, if

$$\left\langle\forall x\in\vec{I},y\in\vec{O}::G.x.y\Rightarrow F.x.y\right\rangle$$

### 2.5.2. Glass Box/Structural Refinement

Let $F:\vec{I}\to\left(\vec{O}\to\mathbb{B}\right)$ be a stream processing function. If there exist stream processing functions $F_i:\vec{I}_{F_i}\to\left(\vec{O}_{F_i}\to\mathbb{B}\right)$ for $i\in[1,n]$ for some $n\in\mathbb{N}$, $n\geq 1$, such that

$$F_1\otimes\ldots\otimes F_n\leq_p F$$

we call $F_1\otimes\ldots\otimes F_n$ a glass box (or, alternatively, structural) refinement of $F$.

#### 2.5.3. Interaction Refinement

Let $F : \vec{I}_F \to (\vec{O}_F \to \mathbb{B})$ , $G : \vec{I}_G \to (\vec{O}_G \to \mathbb{B})$ , $A_1 : \vec{I}_G \to (\vec{I}_F \to \mathbb{B})$ , and $A_2 : \vec{O}_G \to (\vec{O}_F \to \mathbb{B})$ be stream processing functions with $I_G \cap O_F = \varnothing$ and $I_F \cap O_G = \varnothing$. We call $G$ an interaction refinement of $F$, and write $G \leq_i F$, if

$$G; A_2 \leq_p A_1; F$$

## 3. Semantics of Services and SOAs

With the preliminaries of Section 2 in place we now give succinct definitions for the terms *service*, *component* and *Service-Oriented Architecture* (*SOA*).

### 3.1. Services

For a syntactic interface $(I, O)$ we call $Q : \vec{I} \to (\vec{O} \to \mathbb{B})$ *service* (cf. [4]), if $Q$ is causal over $dom.Q$. This models that a service can be partial – it does not have to provide a solution for all possible inputs.

### 3.2. Components

We call a service $Q$ with $dom.Q = \vec{I}$ *component* (cf. [4]). Therefore, components are left-total, i.e. they provide an output for every possible input.

### 3.3. Service-Oriented Architectures (SOAs)

In Section 1.3 we have articulated our intuition behind capturing the core of SOAs as dynamic functional programs. Here, we formalize what dynamic functional programs are.

To that end, we introduce an environment $\Sigma$, which associates with every function name $p \in FID$ a function $\overrightarrow{I^p} \to \wp(\overrightarrow{O^p})$. Thus, $\Sigma$ describes the *environment* under which a given function is interpreted:

$$\Sigma \triangleq \left\{ \left( p_i, \overrightarrow{I^{p_i}} \to \wp(\overrightarrow{O^{p_i}}) \right) : p_i \in FID \right\}$$

A dynamic functional program $RS$, then, consists of the composition of two stream processing functions $r$ and $f$ as illustrated in Figure 3. $r$ models both $RS$'s interface to its environment and the manipulation of $\Sigma$. $f$ expresses the remainder of $RS$'s behavior. $f$ is *subordinate* to $r$ in that only r can communicate directly with $RS$'s environment. If $f$ is subordinate to $r$, then we call $r$ *superordinate* to $f$.

The semantics of a dynamic functional program (or *SOA*) *RS* then arises as follows (with appropriately chosen channel set $C$):

$$\exists \sigma \in \Sigma^\infty : \forall \varphi \in \vec{C}, t \in \mathbb{N} :$$
$$RS.\varphi.t \equiv (\sigma.t.f \otimes \sigma.t.r).(\varphi.t)$$

Intuitively, this means that at any time point, the behavior of $RS$ is given by the composition of $f$ and $r$, as defined in $\Sigma$ *for that time point*. The terms $\sigma\downarrow t.f$ and $\sigma\downarrow t.r$ model *service lookup* and *binding* at time $t$.

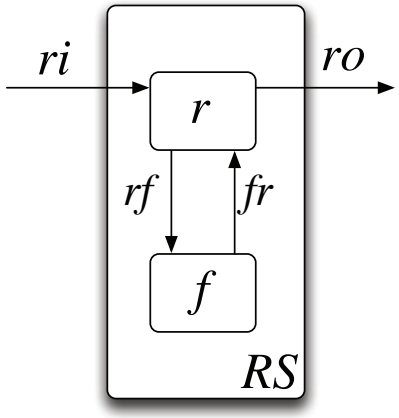


Figure 3: Structure of Dynamic Functional Programs

One way to articulate explicitly how $\Sigma$ changes over time is to view $\Sigma$ as the data state guarded by a stream-processing function, and to compose it directly with $RS$ so that $r$ can manipulate $\Sigma$ via explicit messages to that guarding function. To that end, we model a stream-processing-function factory

$$Q_{\Delta}^{\Sigma} : \left(\Sigma \times \left(O \to M^{*}\right)\right) \to \left(\vec{I} \to \wp\left(\vec{O}\right)\right)$$

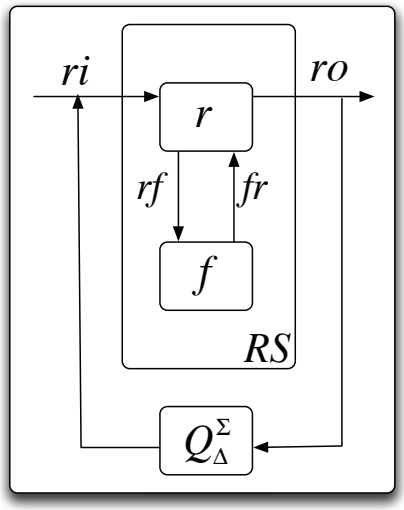


Figure 4: Manipulation of the Environment Modeled as Stream Processing Function

that produces a stream processing function from a given initial state for $\Sigma$, and initial (typically empty) outputs for its output channels.

For simplicity, we identify $Q_\Delta^\Sigma$ with the stream processing function it generates, and obtain $RS \otimes Q_\Delta^\Sigma$ as the semantics for a dynamic functional program where the transformation of $\Sigma$ over time is given by $Q_\Delta^\Sigma$ (cf. Figure 4).

### 3.4. Observations and Discussion

With the definitions in this section we have succinctly captured the essence of SO*: functions express services with interfaces and behaviors, dynamic functional programs capture dynamic function binding.

It is intriguing that we inherit all of the refinement notions defined in Section 2.5: in particular, we can use structural refinement on both $f$ and $r$ to facilitate service modeling at scale. We elaborate this topic in Section 4, where we introduce a specific hierarchical pattern for SO* with practical applications.

Furthermore, we note that we have placed very few restrictions on the functions $f$, $r$, and $Q_\Delta^\Sigma$; specifically, we allow that $r$ (not $f$!) interact with $Q_\Delta^\Sigma$ to manipulate $\Sigma$. This gives $r$ the important role of defining a "policy" by means of which $f$ interacts with its environment, including $Q_\Delta^\Sigma$.

We leave open how the manipulation of $\Sigma$ proceeds in detail. We entertain both of the following scenarios:

(1) $r$ alone manipulates $\Sigma$, i.e. $Q_\Delta^\Sigma$ is subordinate to $RS$.

(2) $Q_\Delta^\Sigma$ has additional input/output channels, allowing the environment to manipulate $\Sigma$ as well.

Each of these scenarios has interesting methodological consequences. (1) gives $r$ complete control over the structure and behavior of the dynamic functional program, including over its own fate. (2), on the other hand, gives the environment control over $RS$'s structure and behavior; this includes the possibility of forcing a replacement of $r$.

We also observe that the semantics as defined already provides for dynamic instantiation of services: We can always replace $f$ at time $t$ with $f \otimes f'$ at time $t+1$ for some suitably chosen $f'$ to model the dynamic instantiation of the function associated with $f'$. To that end, it is useful to view the set $FID$ as containing the infinite universe of all function identifiers, including, for instance structured *names*[3] such as "$f.1.2$" or "$f.g.h$" to indicate name hierarchies. Analogously, we expect $\Sigma$ to contain the infinite universe of all channel names between all function names from $FID$. Whether such a channel is significant at "runtime" is then only determined by the existence of messages on that channel.

Finally, the notion of service lookup and binding we have introduced is purposely rudimentary. We can easily expand it by replacing, say, $\sigma_{\mathbb{1}.f}$ with a

[3] Note that in these names "." is only a syntactic separator between parts of a structured name, and does *not* reflect function composition as elsewhere in this paper.

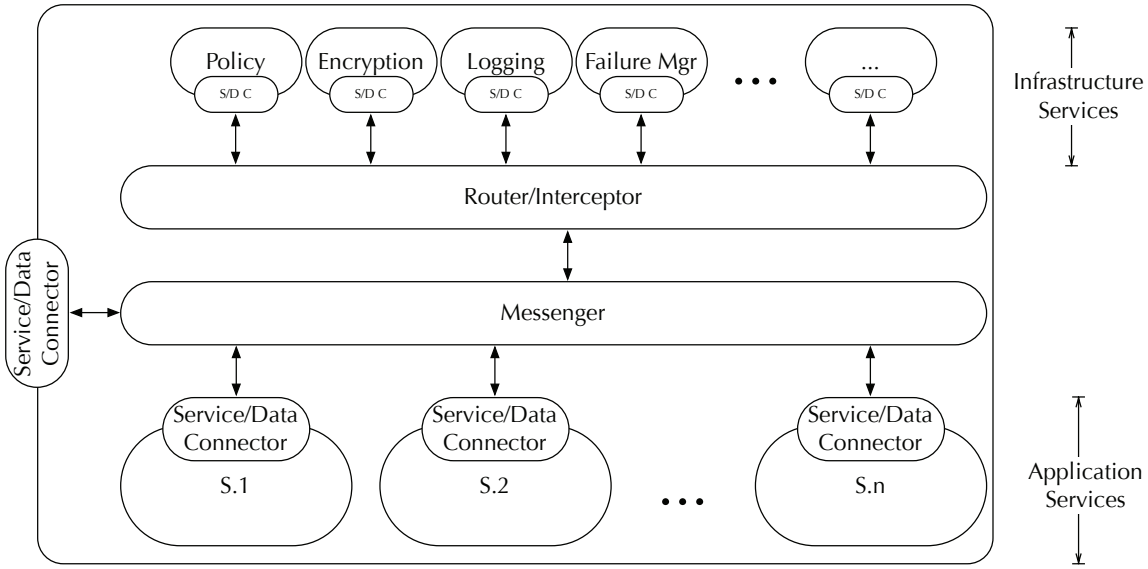


Figure 5: Rich Service

function $lookup(\sigma, t, f_{spec})$ such that $f_{spec}$ is a specification of properties we want the resulting service to have. The name "$f$" is the one and only property we have exploited in the semantics shown here. However, it is a simple exercise to identify a property model for service specifications of which the function name is but one component. This will result in a more structured definition of $\Sigma$'s domain.

## 4. Rich Services: SOAs at Scale

We have identified support for scale as one of the key qualities of SO* in Section 1.3. Now, we introduce *Rich Services* [11], a type of Service Oriented Architecture (SOA) that models a system as an orchestration of loosely coupled services, where services themselves can be decomposed into further Rich Services or implemented as atomic functions. Rich Services form a service hierarchy conforming to a composite pattern [12]. This gives us a handle at addressing scale both in practice and in our formal treatment of SO*.

In essence, a Rich Service is a service as defined in Section 3.1: it transforms one or more streams of input messages into one or more streams of output messages. The utility of Rich Services comes from the structural refinement (cf. Section 2.5) we perform to facilitate:

1. Decoupling of *cross-cutting* or *infrastructure* services from core functionality or *application services*,
2. Loose coupling of its constituent services,
3. Principled control over the interactions among all constituent services,
4. Dynamicity in structure and behavior of the constituent services.

Figure 5 shows the structure of a Rich Service. The Service/Data Connector (SDC) defines the Rich Service's input streams and output streams. The message router transforms the input streams to output streams by orchestrating

interactions between Rich Application Services (RASs) – it passes messages (via a message transport) to RASs, receives RAS output messages, and determines which RAS outputs feed into RAS inputs. Rich Infrastructure Services (RISs) perform functions that crosscut RAS interactions.

Given that the message router mediates RAS-to-RAS interactions, it can also enable crosscutting processing by interposing a RIS into the interaction, replacing the RAS-to-RAS interaction with a RAS-to-RIS-to-RAS interaction. Critically, we use this interposition to enable injection of policy-based constraints and features onto simple, basic workflows, responsive to emerging stakeholder requirements.

### 4.1. Rich Service Semantics

To assign a semantics to a given Rich Service $RS$, we model each of the constituent services $sdc$ (the Service/Data Connector), $msg$ (the Messenger), $ri$ (the Router/Interceptor), and a family $\{is_i\}$ of RISs as services in the sense of Section 3.1, and then find $r$ such that

$$r \leq_p sdc \otimes msg \otimes ri \otimes \left( \bigotimes_i is_i \right)$$

holds and the channel associations respect the ones shown in Figure 5 (i.e. there are no connections among services other than the ones indicated in the figure). Then, we select $f$ such that for a family $\{as_j\}$ of RASs (also services in the sense of Section 3.1) we have

$$f \leq_p \bigotimes_j as_j .$$

Then, the semantics of $RS$ is as defined in Section 3.3:

$$\exists \sigma \in \Sigma^{\infty} : \forall \varphi \in \vec{C}, t \in \mathbb{N} :$$
$$RS\,\varphi\,t \equiv \left( \sigma\,1.f \otimes \sigma\,1\,r \right).(\varphi\,t)$$

## 4.2. Semantics of Hierarchical Rich Services

Both RASs and RISs can, themselves, be Rich Services, thus enabling the

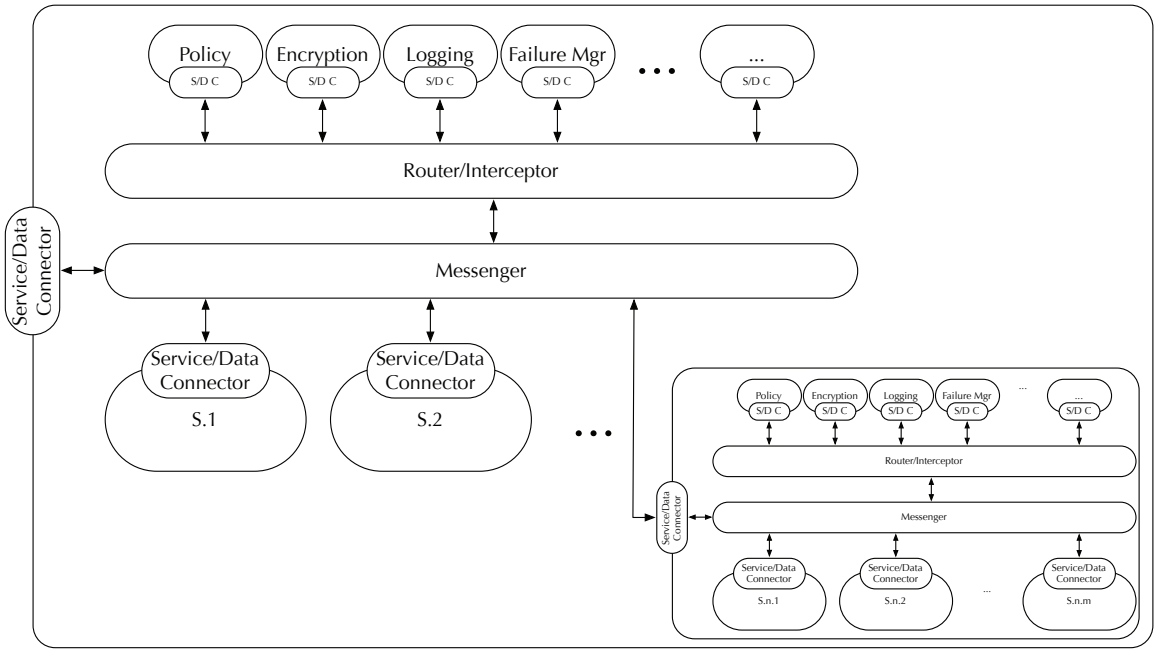


Figure 6: Hierarchical Rich Service

decomposition of the abstractions each represents and completing the composite pattern. This is illustrated for the case of RAS $Sn$ in Figure 6.

We use structural refinement to express hierarchical decomposition. Specifically, we can replace $Sn$ with any family $\{S_k\}$ such that

$$\bigotimes_k S_k \leq_p Sn$$

holds. In particular, this means we can use a decomposition as introduced in

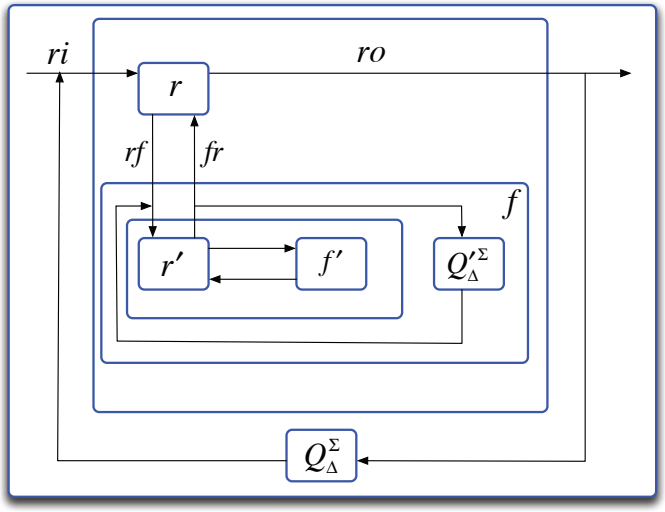


Figure 7: Structural Refinement of *f* in the context of Dynamicity

Section 4.1 to refine $S\!n$ into another Rich Service.

It is instructive to look at the decomposition of a Rich Service $RS$ consisting of functions $r$ and $f$, such that $f$ gets structurally refined by means of functions $f'$ and $r'$.

Figure 7[4] illustrates this situation, including the relationship to the environment $\Sigma$. In particular, $r$ and $r'$ are connected via channels as $f'$ is subordinate to $r'$. Furthermore, $f'$s decomposition can have its own $\Sigma$ manipulator $Q'^{\Sigma}_{\Delta}$, which performs dynamic binding changes relating to $r'$ and $f'$ only.

This decomposition can, of course, analogously occur for $r$.

### 4.3. Observations and Discussion

The chosen formalization for Rich Services is fully generic with respect to the roles played by $sdc$, $msg$, and $ri$. *Together* (and with their connected RISs) they are modeled by $r$. This leaves a lot of methodological room for interpreting these roles, and for moving responsibilities between them. Obviously, $ri$ is intended to support routing and interception of messages communicated via $msg$. Because the semantics of Rich Services is dynamic in the interpretation of $ri$ (in particular), we can use updates to $ri$ to dynamically change the routing of messages within any and all services subsumed by that Rich Service.

Note that typical Enterprise Service Bus (ESB[5]) systems rely on a decomposition similar to what Rich Services provide; hence, the chosen formalization can be used to articulate a semantics for ESBs as well. In these ESBs it is typically possible to insert routers and transformers into the message flow between two services – we can model this very directly by means of corresponding decompositions and interpositions of $ri$ and the RISs. For instance, by using relations such as $ri \leq_p ri_1 \parallel ri_2$ or $ri \leq_p ri_1; ri_2$ we can model parallel and sequential flows in $ri$ for appropriately chosen $ri_1$ and $ri_2$ in addition to the choice that is inherent in the set-valued nature of stream processing functions. This gives us a rich language for expressing routes within a Rich Service. Other than most ESBs we can dynamically change the routing and also even the constituent services in response to message flow.

The selection and interposition of RISs into regular service conversations (message exchanges) allows addressing of a vast set of infrastructure concerns. Obvious, and frequently cited, examples are logging, encryption/decryption, as well as data transformations. Other examples include failure management and management of general policies, such as HIPAA. For failure management, the router might deliver a message to multiple different copies of a service, receive their responses, and perform a majority-vote on the results. For HIPAA, a policy

[4] For better readability, boundaries of services are shown in blue in the figure.

[5] ESBs come in a wide variety of implementations with varying feature sets; for two examples we refer the reader to [36] and [35].

RIS could interpose via *ri* to ensure personally identifying information is removed from patient data before its delivery to the environment. Clearly, there are many more sophisticated scenarios than these, but they help illustrate the utility of the Rich Services pattern in practice.

The chosen formalization of a Rich Service $RS$ via the composition of $r$ and $f$ clearly articulates the role of $r$ as the "interposer" between any interactions generated by (any refinement of) $f$. This emphasizes $r$'s capacity to enforce policy as it pertains to $f$ (and its refinements). Specifically, we can now formally define what a policy *is* and *does* in a Rich Service. Let $f$ and $r$ be stream processing functions. We call $r$ a *policy* with respect to $f$ if $r$ is superordinate to $f$.

A nontrivial policy, where $r$ differs from the identity function, modifies $f$'s behavior, i.e. $r \otimes f \neq f$. If we have $\langle \forall \varphi \in \vec{C} :: (r \otimes f).\varphi \neq f.\varphi \rangle$, then $r$ modifies all behaviors of $f$. This may occur if $f$ alone has the potential to harm its environment, and $r$ expresses a "healing" policy so that $f$'s effects are mitigated. If we have $r \otimes f = r$, then $f$ is irrelevant, and policy $r$ subsumes $f$. If we have $r \otimes f \leq_p f$, then policy $r$ constrains $f$'s freedom in choice of behaviors – this is the classical role that policies play in programming constructs such as "if then else". Clearly, there is significant methodological potential in the interplay between a policy $r$ and its subordinate function $f$; we will discuss this further in a forthcoming paper.

## 5. Open Rich Services (ORS): A DSL for SOA at Scale

In the preceding two sections we have introduced a concise formal definition for services and SOAs, and have shown the utility of these definitions also for systems that scale hierarchically while decoupling application from infrastructure concerns. The resulting Rich Service pattern is being applied in a broad range of software and systems engineering and integration projects, in domains such as Ocean Sciences [13], Health Sciences [14], and Automotive/Avionics [15]. In this section, we show the utility of the Rich Service pattern using a realistic example from the CitiSense[6] project currently underway at UCSD. To that end, we also present a domain specific language (DSL) for Rich Services, called Open Rich Services (ORS). ORS, in particular, supports late- and dynamic binding of service names to service functions, and allows service functions to be specified using the Open Source functional programming language Clojure. This closes the loop between our formal definitions for services and SOAs, and the notion of dynamic functional programs.

In Section 5.1 we introduce the example, "CellSense". In Section 5.2 we briefly discuss what makes CellSense challenging to design and implement. In Section

[6] http://citisense.ucsd.edu

5.3 we introduce the ORS DSL and its runtime system in Clojure. Our presentation here largely follows [16], where we have first described CellSense and the Clojure-based DSL; specifically, we add a discussion of the relationship to the formal semantics throughout this section.

### 5.1. Example: CellSense

The example we develop in this section crystallizes many concerns of SO*, ranging from functional separation of concerns, to architectural reuse, to dynamic service lookup and binding. While the example may seem mundane at first, it has many intricacies, especially when contemplating multiplicities and dynamic update of its constituent parts.

Consider a system ("CellSense", cf. Figure 8) consisting of a cell phone with a display, a data store, and a processing capability – each internal to the cell phone. Attached to the cell phone is a sensor, say, for monitoring CO levels. Furthermore, there is a remote data store, and a remote processing capability. The basic behavior of the system shall be that sensor values are stored and, in parallel to storing the value, a derived value is computed and subsequently displayed. If we informally represent capabilities or "functions" by words,

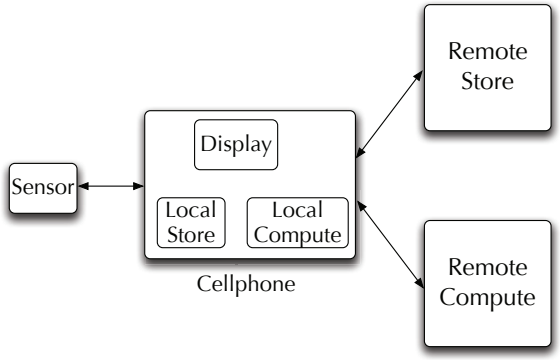


Figure 8: CellSense Basic Architecture

sequencing by juxtaposition, and parallel execution by a prefix "||", respectively, we can describe the basic behavior of the cell phone below (with parentheses to group elements of the specification). We label it and refer to it throughout as "(*)".

(*) sense (|| store (process display))

Assume further that the system is supposed to fulfill a number of additional requirements, listed as rules in Figure 9. These rules refer to the cell's and the sensor's power level as CPL and SPL, respectively.

| # | Rule |
|---|---|
| 1a | CPL high: store locally and remotely, compute locally |
| 1b | CPL low: store locally only, compute remotely |
| 1c | CPL very low: stop storing, computing and displaying |
| 2a | SPL high: provide value every second |
| 2b | SPL low: provide value every minute |
| 2c | SPL very low: provide value every hour |
| 3 | Encrypt/decrypt remote messages |
| 4 | Once per day, if CPL is at least low, bulk-transfer all local data to remote store |
| 5 | The location for the remote processing capability can be set at system startup and changed dynamically. |
| 6 | The number of cell phones, and sensors per cell phone, as well as the numbers of remote stores and compute facilities is unbounded and can change over time. |

**Figure 9: CellSense Requirements/Rules**

CellSense is representative of a large class of distributed, reactive, and dynamic systems. This class includes, for instance, sensor/actuator networks, observatories, and other cyber-physical systems. The specifics of CellSense were extracted from CitiSense, a system for community-driven behavioral and environmental health monitoring currently under development at UCSD. CitiSense presents many challenges that are representative of complex systems of systems, as well as of SO*-attempts at addressing them. These challenges range from energy-management to observation of privacy policies, to fault-tolerance, to highly diverse resource capabilities of devices that participate in the infrastructure to the need for elasticity and scalability in the number of participants, sensors/actuators, computation, storage and networking capabilities. Most of these fall under the requirements categories introduced above; others, such as resource and deployment diversity deserve additional attention and are outside the scope of the current paper.

We will refer to CellSense requirements solely by their number – "requirement (1)" and "(1)" are interchangeable. Furthermore, by (1) and (2) we refer to requirements (1a) through (1c), and (2a) through (2c), respectively.

We can understand rule (1) as defining a policy for where computation and storage are to occur relative to available power for the cell phone. Similarly, rule (2) sets a policy for the frequency by which sensors deliver readings, relative to available power for the sensor. Rule (3) establishes a security policy. Rule (4) establishes an information assurance (backup) policy relative to available power for the cell phone. Rules (5) and (6) together demand dynamicity and scalability.

### 5.2. What makes CellSense Challenging?

CellSense brings forward a number of concerns worth discussing in more detail. First, requirements (1) and (2) refer only to the cell phone and the sensor, respectively. Yet, they constrain *all* behaviors of each. Solutions that *entangle* (*), (1) and (2) will become brittle if any one concern were to change.

What does entanglement mean in this context? Consider where and how you would implement the logic for (*), (1) and (2), respectively.

A classic Object-Oriented (OO) design would introduce classes for the various entities (or actors) in the system. We might end up with classes for the Sensor, the CellPhone, the Store (LocalStore and RemoteStore might be obvious subclasses), and the ComputeFacility (LocalCompute and RemoteCompute might again be obvious subclasses.) We might then identify corresponding methods, such as "store", "process", "display" and "sense"; similarly attributes and access methods can model the power levels.

This structural and behavioral decomposition has utility until we arrive at the implementation of the rules; at that point we realize that the rules really are cross-cutting in nature.

At a high level, (*) describes the cell phone's behavior overall. (1) and (2) express that CPLs and SPLs need to be monitored, and depending on the monitored value, multiple modifications to (*) are necessary. For instance, the CPL has an influence on all of storing, computing and displaying. Clearly, we could implement this with conditionals in each of the implementations of *store*, *process*, and *display* in the respective classes identified above. Similarly, the SPL could result in conditionals within the implementation of *sense* – which would be contained either within the sensor's or the cell phone's implementation, depending on whether sensor readings are pushed or pulled, respectively. Now, if either one of (*), (1), or (2) were to change, we would have to go back to all of the implementations separately to effect and validate the proper change. While this is manageable within this small-scale example, this effort becomes quickly intractable for any larger scale.

Second, (1) *influences* the behavior (*); it does not request a change in the general *behavior pattern*. Under the entangled implementation described above, the general behavior pattern is obscured by the conditional logic to support (1) – even in this small example this becomes distracting, due to the requirements' simultaneous impact on multiple steps of the behavior pattern.

Third, (2) and (4) both demand a "scheduling" capability, which signifies another common behavior pattern that repeats across system components, despite the obvious differences between eventual implementation and deployment environments.

Fourth, the behavioral influence of (3) is tied to the constraints imposed onto (*) via (1) and (4). Therefore, neither an entanglement with (*) nor with each of (1) or (4) is desirable.

Fifth, requirement (5) implies the ability to indicate, and modify at runtime, *where* a particular system capability is to be deployed.

Sixth, requirement (6) implies the ability to dynamically instantiate and remove instances of any one of the system parts shown in Figure 8.

In summary, while there are many different ways to design and implement CellSense, SO*-based approaches hold promise in addressing the distribution (logical and physical) and dynamicity adequately and flexibly. In addition,

CellSense benefits from a particular SO*-style in which infrastructure concerns are cleanly separated from application concerns, which further facilitates reuse.

A broad range of design patterns exists for OO that provide some relief in terms of the entangling that a naïve design would result in. We mention the Strategy, Template Method, and Composite patterns [12] explicitly, which can be fortuitously combined to factor the mentioned concerns in a more manageable way. Whether any concrete combination of these patterns does provide the intended benefits largely depends on the individual skill set of the modeler. ORS institutionalizes the combined benefits of these patterns in a reusable desing.

We will now show how to model these requirements succinctly in ORS.

## 5.3. Open Rich Services DSL in Clojure

Expressing a system like CellSense using stream processing functions directly is certainly possible, albeit laborious. Therefore, we have developed an Architecture Definition Language (ADL) in the form of an embedded domain specific language (DSL) for ORS. As an embedded DSL, an ORS specification *exists as part of the executable code at runtime* within the host language and its runtime system. This opens significant opportunities for runtime checking and principled evolution of structure and behavior at runtime.

### 5.3.1. Clojure

As the host language we have chosen Clojure [17] [18], a new LISP on the Java Virtual Machine (JVM) with immutable, highly performant data structures [19] (including lazy lists, key/value maps, vectors and sets). Clojure is a non-pure functional programming language with explicit in-process concurrency support via a software-transactional memory. This combination allows us to leverage Clojure's functional fragment to model services as functions, and service composition as the composition of functions. Furthermore, we can use Clojure to specify structure and behavior to any desired level of detail – we have the full power of a complete programming language at our disposal when necessary, while being able to articulate architecture specifications at a high level as well.

Clojure's code-as-data philosophy inherited from LISP provides significant flexibility in defining the DSL in anticipation of its use as a tool for model-driven engineering. Because Clojure embraces the JVM, the DSL inherits excellent interoperability features with Java libraries. The interoperability goes both ways: any Clojure module can be compiled such that it can be called from Java programs.

### 5.3.2. Basic Concepts: Services as Functions

A basic Open Rich Service (ORS) is a Clojure function taking a message as input and producing a message as output. The following example shows a "basic-compute" ORS in Clojure – it simply returns the message it receives.

```
(defn basic-compute [rsm] rsm)
```

In Clojure, we use `(defn f [args] bdy)` to define function *f*, whose formal parameters are denoted by *args*, via the body *bdy*. In the example above, *basic-compute* is the function name, *rsm* is the name of the argument, and the body consists of returning the argument unchanged.

This is clearly a restricted form of the stream processing functions we have introduced in Section 2.4. Specifically, `basic-compute` has only access to the message sent at a given time point instead of the full "channel" history. In this sense, we can think of `basic-compute` as a state machine encoded in Clojure, whose canonical mapping into the realm of stream processing functions would occur as we have hinted at in Section 3.3 for the environment Σ. In Section 5.5 we discuss how we can expand the pure functional view via persistent message histories so that Clojure service functions can indeed access their past message history.

We also note that the DSL has no explicit channel concept; it is implicitly there, of course, via the Clojure-defined wiring of composite functions.

Furthermore, in this service definition, the type of *rsm* is left unspecified – any Clojure type will do. In particular, we can use a Clojure key/value map, and use the keys to structure the message type we hand to services. For instance, we could use a map

```
{:type "basic-message"
 :r-value "hello world"}
```

to indicate a specific message type "basic-message", and its content value "hello world"; the basic services could then dispatch further on the message type, or process the `:r-value` while constructing their own return message.

The genericity of key/value maps works in our favor here, because we can augment the messages we pass around according to both infrastructure and application needs.

### 5.3.3. Composite Services

A composite ORS also maps input messages to output messages. However, it does so by coordinating any number of its operand ORSs (which are, in turn, either basic or composite). Coordination happens by routing messages from one service to another. Therefore, a composite ORS comes equipped with a specification of the message flow among its constituent services.

Figure 10 shows an example of a composite ORS. Here, we define *cell-phone* as a composite, which coordinates a number of basic services named *:compute*, *:store*, and *:display* (among others). The actual coordination is specified as the *:sense* service. This represents the function to be executed when a new sensor value is available. When *:sense* is called, in parallel, the *:store* function and the sequential composition of *:compute* and :display are executed. In ORS, parallel composition

is denoted by a prefix "||" operator[7]; sequential composition is indicated by the **compose** function, whose argument is the sequence of function names to be applied in the given order. ORS specifies *bindings* between names and an implementation. For instance, in Figure 10 we specify a binding between the name *:compute* and the implementation (the Clojure function from above) *basic-compute*. This binding can be specified and changed at a later time – even dynamically at runtime.

Thus, we use Clojure's innate ability to bind names to functions, as well as the explicit **bind** function to express the initial environment $\Sigma$, as well as changes to $\Sigma$ over time. For service lookup we rely on Clojure's built-in function lookup, as well as on an explicit registry, which we describe in more detail in Section 5.4 This is an implementation strategy for the binding as required by the semantics given in Sections 3.3, 4.1 and 4.2.

The services of a composite RS each fall into one of two categories: *application* and *infrastructure services*. Application services define the application logic. Infrastructure services, on the other hand, *manipulate* application and data flow, or provide supporting tasks to that flow. The example in Figure 10 introduces multiple infrastructure services including *:encrypt*, *:decrypt*, and *:transfer*. The first two infrastructure services are again modeled as basic services specified elsewhere. *:transfer* uses a predefined infrastructure service *schedule*, which takes a frequency as its first argument, and a service to be executed according to that frequency as its second argument. The meaning of this specification is that when the *cell-phone* ORS is instantiated (see next subsection), the *:transfer-data* function is scheduled to execute once per day (at the discretion of the system's scheduler. This addresses requirement (4).

The final element to specifying a composite RS is to declare *how* the infrastructure services manipulate or support the application services. To that end, we introduce the notion of *transforms*. A transform takes a given service (application or infrastructure), and places it in relation with another infrastructure service.

The use of transforms is an implementation strategy for the routing and interception mechanism as defined by the composition

$$sdc \otimes msg \otimes ri \otimes \left( \bigotimes_i is_i \right)$$

from Section 4.1. Specifically, as we will see, below, transforms express bindings (i.e. updates to the environment $\Sigma$) as well as routing of messages among across $msg$ and the family $\{is_i\}$ of infrastructure services.

[7] In LISPs, we denote a call to function *f* with argument list *a1* through *an* by prefix notation within parens as ***(****f a1 … an****)***

Following the introduction of Rich Services in Section 4, we have to specify:

1. Rich Services (the container)
2. Application Services
3. Infrastructure Services & The Router/Interceptor
4. The Messenger

```
(def cell-phone
  (rich-service
    (app-services
      :sense
        (|| :store
          (compose :compute :display)),

      :ls             local-store,
      :lc             local-compute,
      :display        display,

      :power-high     high-power-test,
      :power-low      low-power-test,
      :power-vlow     vlow-power-test,
      :store          basic-store,
      :compute        basic-compute,
      :rs             remote-store,
      :rc             remote-compute,
      :transfer-data  transfer-data)

    (infra-services
      :encrypt        encrypt,
      :decrypt        decrypt)

    (transforms
      :rs (pre :encrypt),
      :rc (pre-post :encrypt :decrypt),
      :store (bind
               (cond-flow
                 :power-high  (|| :ls :rs),
                 :power-low   :ls,
                 :power-vlow  :skip)),
      :compute (bind
                 (cond-flow
                   :power-high  :lc,
                   :power-low   :rc,
                   :power-vlow  :skip)))

    (schedules
      :transfer (every :Day
                  (cond-flow
                    (not :power-vlow)
                      :transfer-data)))))
```

Figure 10: CellPhone Rich Service in ORS DSL

5. The Service/Data Connector.

We will now introduce the DSL concepts for each of these elements, in turn.

### Rich Services

In ORS we define Rich Services via a call to the **`(rich-service …)`** function. Its result is a Clojure map storing the infrastructure and application services defined *within* this Rich Service.

In Figure 10 we define a Rich Service for the cell phone as follows:

```
(def cell-phone
  (rich-service
    (app-services …)
    (infra-services …)
    (transforms …)
    (schedules …)))
```

Each of the `app-services`, `infra-services`, `transforms` and `schedules` functions return a correspondingly labeled Clojure map capturing the service definitions given as arguments to the respective function.

### Application Services

In ORS we define Application Services via the **`(app-services …)`** function. Its argument sequence consists of pairs of the form `keyword service-definition`**.** The service definition is either a reference to a service function (basic or Rich Service), or an expression defining the flow among multiple existing services. The keyword defines how that service definition is know within the enclosing Rich Service.

In the example of Figure 10 the application services definition begins as follows:

```
(app-services
  :sense
    (|| :store
      (compose :compute :display)),

  :ls             local-store,
  :lc             local-compute,
  :display        display, … )
```

This introduces the name (i.e. Clojure keyword) `:sense` for the expression **`(||`** `:store` **`(compose`** `:compute :display`**`))`**. As explained above, this definition expresses the *parallel* composition of the function known as `:store`, and the expression **`(compose`** `:compute :display`**`)`**. The latter indicates the *sequential* composition of the services known as `:compute` and `:display`, respectively.

Therefore, `:sense` defines a service function that routes an incoming message, in parallel, to `:store` and the implicitly defined service function for **`(compose`** `:compute :display`**`)`**. Its result is a message that contains the return values from *both* of its operand services.

The keywords `:ls`, `:lc`, and `:display` refer directly to (i.e. introduce shorthand notation for) the service functions `local-store`, `local-compute` and `display`, respectively. These keywords can occur in service expressions.

### *Infrastructure Services & The Router/Interceptor*

Rich Services, as introduced above, articulate two layers of service interactions. First, they express interactions among the application services as direct calls among them, or via service expressions as demonstrated for application services in the previous paragraph. Second, they express *infrastructure* workflows that capture cross-cutting aspects that are imposed upon the interactions among the application services.

In essence, the interactions among application services define message routes; the interceptor establishes bindings between triggering messages (or message patterns) and resulting behaviors (services to be executed when the trigger has occurred.) Often, this will include execution of an infrastructure service before a (modified) message is relayed to the intended (or a substitute) recipient. This establishes a tight interplay between the router/interceptor, and the infrastructure services.

In the following, we address how we specify routes, triggers and responses in ORS; this captures a significant subset of the generic form of intercepted service routing discussed for Rich Services in general.

In ORS we capture these infrastructure workflows by (a) listing the infrastructure services, and (b) by specifying the interplay between the infrastructure services and the application-level workflows. To accommodate (a) we introduce an `(infra-services …)` call into the `(rich-service …)` specification, which defines keyword-shorthands for service names. To accommodate (b), we introduce a corresponding `(transforms …)` call, which specifies how calls to services, or results of service calls shall be modified via other interposed service calls.

In the Example of Figure 10, we introduce the infrastructure services via a call to

```
(infra-services
  :encrypt encrypt,
  :decrypt decrypt)
```

Furthermore, we specify routing and interception via the `(transforms …)` call:

```
(transforms
      :rs (pre :encrypt),
      :rc (pre-post :encrypt :decrypt),
      :store (bind
               (cond-flow
                 :power-high  (|| :ls :rs),
                 :power-low   :ls,
                 :power-vlow  :skip)),
      :compute (bind
                 (cond-flow
                   :power-high  :lc,
```

```
                        :power-low   :rc,
                        :power-vlow  :skip)))
```

The transforms specification consists of pairs of the form keyword/binding-transform. The keyword indicates a service name, the binding-transform determines what should happen if and when that service is called (or, in other words, when a message is routed to that service).

The pair `:s1 (pre :s2)` determines that whenever a message `m` is routed to `:s1`, it is actually *first* routed to `:s2`, and the result of `:s2` is then routed into `:s1`.

Similarly, the pair `:s1 (post :s2)` determines that whenever a message `m` is routed to `:s1`, the result of `:s1` is then routed to `:s2`.

The pair `:s1 (pre-post :s2 :s3)` determines that whenever a message `m` is routed to `:s1`, it is actually first routed to `:s2`, and the result of `:s2` is then routed into `:s1`. In turn, the result of `:s1` is then routed to `:s3`.

We use the pre-binding to ensure that any calls to the remote store (`:rs`) are encrypted prior to a message leaving for `:rs`. We use the pre-post-binding to ensure that messages for the remote compute-facility are encrypted before they hit `:rc`, and the response from `:rc` is decrypted before it arrives at the cell-phone.

Using the router/interceptor mechanism we accomplish a disentanglement of the calls relating to encryption and decryption from the application-level workflow specified by the `:sense` service.

Another powerful routing specification tool ORS provides is the `:keyword (bind …)` pair. It allows us to replace the existing binding of a service name with another, for instance one that reroutes the message from one service to any other.

In Figure 10 we use this to dynamically bind the `:store` service, depending on the state of the system, or on the content of messages received prior:

```
  :store (bind
           (cond-flow
             :power-high  (|| :ls :rs)
             :power-low   :ls
             :power-vlow  :skip))
```

In this example, we use the `(cond-flow …)` function ORS provides to substitute the appropriate service expression for the call to the `:store` service. `(cond-flow :s11 exp12, :s21 exp22, …, :si1 expi2 …)` routes the incoming message first to the service `:s11`; if the result of that service is *true*, the *original* message will be routed to the service expression `exp12`, and its result is the result of the `(cond-flow …)` call. Otherwise, i.e. if `:s11`'s result is *false*, the message is routed to `:s21`, repeating the pattern. In general, a message is routed to the first `expi2` for which `:si1` evaluates to *true*; the result of `(cond-flow …)` is then the result of that expression `expi2`. If none of the `:si1` evaluate to

true, `(cond-flow …)` is equivalent to the special service `:skip`, which simply returns any incoming message.

In the example, we dispatch the incoming message based on the power-state of the cell-phone as determined by the service calls to `:power-high`, `:power-low` and `:power-vlow`, respectively.

Additionally, an ORS `(rich-service …)` specification can contain `(schedules …)` calls, which accept pairs of the form `keyword (every …)`. The keyword denotes a service, and the `(every …)` call determines the frequency with which the specified service is being called.

In Figure 10, this call looks as follows:

```
(schedules
      :transfer (every :Day
                  (cond-flow
                    (not :power-vlow)
                      :transfer-data)))))
```

Here, the :transfer service is bound to the

```
(cond-flow
    (not :power-vlow)
      :transfer-data))
```

expression, and executed daily at the time the cell-phone service was deployed and started.

The functions we have introduced in the previous paragraphs significantly help in specifying the routing and interception of messages, and thus provide *one* way to express these key elements of the Rich Services pattern. Clearly, this is not the only way, nor complete relative to what the general Rich Services pattern enables. However, the combination of pre/post and bind transforms, conditional, sequential and parallel flows, and schedules covers a broad spectrum of desirable dynamic bindings.

### *The Messenger*

In Rich Services in general, the role of the messenger is to provide a communication mechanism among the constituent services. In ORS we use Clojure function calls as the main *internal* conveyance of messages among services. Specifically, within the same JVM, messages travel via a sequence of function calls from one service to the next. Externally, i.e. between services deployed among multiple JVMs, we use http coupled with corresponding web servers as the transport for ORS messages. We discuss the details of this mechanism in Section 5.4.

### *The Service/Data Connector*

The service/data connector is a Rich Service's external interface. In ORS this interface is determined by the application services defined for a given `(rich-service …)` call. Its enforcement is described, in detail, in Section 5.4.

### 5.3.4. Service Invocation

So far we have shown how to define basic and composite services, and how to invoke basic services from within composite ones. Now we demonstrate how to invoke another composite service. To that end, consider the following segment of the `sensor` Rich Service:

```
(def sensor
  (rich-service
    (app-services
      :val (fn [rsm] ...)
      :push (compose :val :cell-phone/sense))
    (schedules
      :delivery-second
        (every :Second (cond-flow (:s-power-high) :push))
      :delivery-minute
        (every :Minute (cond-flow (:s-power-low) :push))
      :delivery-hour
        (every :Hour (cond-flow (:s-power-vlow) :push)))))
```

The call `:cell-phone/sense` indicates the service name (`:cell-phone`) before the "/", followed by the application service to be called (`sense`). Recall that ORS uses late binding: resolution of the call to `sense` will be attempted, at runtime, first on the node on which `sensor` is deployed (see Section 5.3.5); if an instance of the `cell-phone` Rich Service is deployed there, it will receive the call. Otherwise, the call will be forwarded to another node (see Section 5.3.5) housing such an instance. If the call cannot be resolved, an exception occurs. This example addresses requirement (2), analogous to `cell-phone`'s conditional flow.

### 5.3.5. Deployment Architecture

The deployment architecture determines (a) how many separate *nodes* (locations of computation) exist in the system, (b) what instances of an ORS will be created, and (c) on which nodes these instances will run.

The following example shows how we deploy the `cell-phone` and `sensor` services. Of each service we deploy a separate instance on each of the two nodes.

```
(rs-start)
(let [[n1 n2] (launch-nodes 2)]
  (deploy-instance n1 :cell-phone1 "examples.adl1/cell-phone")
  (deploy-instance n1 :sensor1 "examples.adl1/sensor")
  (deploy-instance n2 :cell-phone2 "examples.adl1/cell-phone")
  (deploy-instance n2 :sensor2 "examples.adl1/sensor"))
```

The built-in **`rs-start`** DSL command starts the ORS system on the node on which it is executed. We describe the details of this startup process in Section 5.4. The **`launch-nodes`** DSL command starts *additional* nodes – two in this case, bound to the variables `n1` and `n2`, respectively. Finally, **`deploy-instance`** addresses requirement (5) by creating service instances and deploying them on a given node. To that end, we specify the node on which to deploy, the name of the

new instance, and the service specification relative to the Clojure namespace in which it is defined.

The ORS facilities for specifying a deployment architecture help establish the initial state of the environment $\Sigma$ in the sense of Section 3.3, and thus form the starting point for the dynamic bindings expressed via $\Sigma^{\infty}$. In the semantics definition of Sections 3 and 4 we do not model deployment concepts explicitly; they become relevant only in settings where ORSs get executed, which is the topic of the next section.

### 5.4. The ORS Runtime System

In the following subsections, we highlight the direct relationship between services and functions, and the use of dynamic binding to yield SOAs via the ORS DSL. Furthermore, the embedding of the DSL within Clojure makes the DSL specification part of the executing system, as an artifact that can be inspected and modified even at runtime. While we do not exploit this directly in this paper, this is of significance in building introspective analysis and modification of the executing architecture, which we will address in forthcoming papers.

The ORS Runtime system forms an execution context in the sense of Section 1.2.

The basic deployment environment for ORS is the Java Virtual Machine (JVM), with Clojure as the actual execution environment. We refer to the physical machine on which the JVM runs as the *host computer*, or *host* for short (cf. Figure 11). We refer to an instance of the Clojure system running on a specific JVM as a *node*. The node from which the system is *launched* is referred to as the *root-node*. Launching the system consists of loading a file containing an ORS DSL specification.

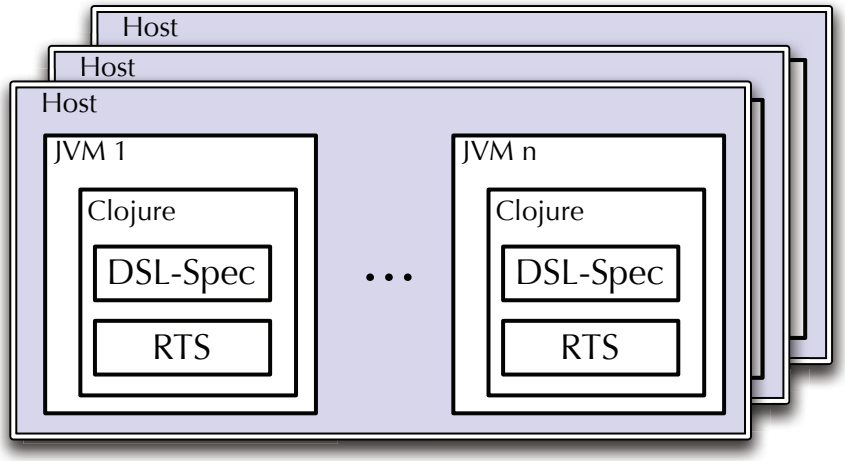


Figure 11: ORS Deployment on Multiple Hosts

Each DSL specification is executed in the context of a Run Time System (RTS) implemented in Clojure. This RTS provides implementations of all the built-in services, such as *skip*, *schedule*, *cond-flow*, `||`, and *bind*, as well as the other transforms. It also provides the facilities for registering/binding and rebinding services to names. Each node has its own *registrar*, which is a data structure and

associated functionality to store bindings between service names and the Clojure functions to be executed when that service is called. The *root-node*, by default, houses the *master registrar*, which serves to resolve cross-node service calls. All service names within the RTS are represented as Universal Resource Identifiers (URIs). Furthermore, the RTS maintains data structures that keep track of the launched nodes so they can be safely shut down when no longer needed.

The startup sequence for an ORS system is as follows:

1. The Clojure system is brought up on the host as a Clojure Read-Eval-Print-Loop (REPL).
2. The Clojure system loads the DSL specification. At this point, the RTS is instantiated, and it proceeds to parse the DSL specification, identifying all application and infrastructure services, transforms, and scheduled services. Each service definition is wrapped with a *service controller*, a Clojure function that performs pre-/post-processing of incoming and outgoing messages, via supplied hooks (plugin-points) for adding pre and post transforms (see Section 5.3). Then, the transforms are applied to the functions representing the application services and the resulting functions are registered with the node's registrar.
3. The deployment information in the ORS specification is parsed and executed. On the root node, all services contained in the specification are registered with the master registrar. If the ORS specification starts additional nodes, the same startup sequence is executed on these nodes, with the exception that their registrars, in addition, register with the *master registrar* of the root-node. Each node is started with a JAR file containing the RTS, including the DSL specification.
4. Fourth, scheduled tasks are launched on the respective nodes. At this point, all services defined by an ORS specification are accessible to the node's environment.

## 5.5. Observations and Discussion

The presented executable ORS DSL operates with the same abstractions as we have introduced them for services and SOAs in Section 3. Directly, we implement services as functions, and SOAs (in the form of Rich Services) as dynamic functional programs.

The ORS DSL clearly makes design decisions to facilitate realizability of the DSL specifications: individual services receive only one message at a time, so that each service specification is akin to a state machine implemented in Clojure. However, because Clojure is a general purpose programming language, service implementations can keep a history of the messages they have received *and* sent so far, allowing service-implementing functions to act as full stream processing functions (at least on finite prefixes of their in- and output streams).

Also, we model channel names only implicitly via the parameter positions in service functions, and rely on Clojure's function wiring and our own service registrars together with the message transport to play the role of channels from our semantics definition. This could easily be expanded via an explicit modeling of a distributed messaging capability with explicit channel names. Of course, this

would have a design impact also on how the base services receive and access their messages, and how they can send their responses.

The bind, pre- and post-transforms cover a significant set of routing and policy definition scenarios, but are by no means as comprehensive as the completely general router/interceptor concept of our semantics definition for Rich Services. Expanding the set of transforms in that direction is future work.

Clearly, there are many other possible ways to implement the presented DSL, and yet more ways for refining the DSL itself – however, even in its current form, the ORS DSL shows how to use SOA concepts to express dynamic architectures, and to exploit reuse across a broad range of subsystems in a SOA. As shown, the ORS DSL code covers all of the requirements specified for Cell Sense in Section 5.1 – while factoring out cross-cutting concerns and avoiding repetitive architecture specifications across sensors and cell phones.

We have yet to fully exploit the possibilities arising from having the DSL specification as an executable part of the runtime system – this, too, is an area of future work.

## 6. Discussion and Related Work

In the preceding sections we have introduced a concise mathematical model for services and SOAs, and have shown its utility as an underpinning for a DSL that captures key concepts of the scalable Rich Services architecture pattern. Here, we discuss its benefits in the context of related work, and potentials for further improvement.

**Rich Services and the OASIS Reference Model** Our introduction of Rich Services and their semantics in this document directly addresses and makes precise the key concepts of the OASIS Reference Model for SOAs [3] as articulated in Section 1.2. *Services* are functions with both syntactic and semantic *interfaces*. *Visibility* is explicitly controlled through both the service interface *and* through the service registry (called *environment* in Section 3.3). The *semantics* of services, messages and SOAs is explicitly defined in Sections 3 and 4, including the *real world effects* on component states and channel contents. These sections also define the abstract *execution context*, whereas Section 5 articulates a concrete execution context in the form of a prototypical DSL implementation for Rich Services. Functions are *service providers* and *service consumers*, *interacting* via messages. Service interfaces act as *service descriptions*. *Contracts* and *policies* are handled via the decoupling of the functional and cross-cutting aspects of a Rich Service, expressed via the decomposition into functions $f$ and $r$, respectively. Therefore, Rich Services as presented do constitute a SOA according to the OASIS Reference Model.

**Concise Semantics for Rich Services** The semantics definition is concise because of our choice of streams and stream processing functions as the formal underpinning. While we share the static notions of services and components with our preliminary work in [4], we have focused here on modeling dynamic binding as *the* core SOA concept besides the service notion itself. We modeled

service bindings via an environment that evolves over time. As a consequence, manipulation of this environment becomes a necessity worthy of explicit modeling – we chose to let the SOA manipulate its binding environment, but other choices are conceivable as well.

Clearly, there are other options we could have chosen as the basis of our semantics. Principal alternatives to streams and stream processing functions as underlying semantic foundation are, for instance, process algebras such as CSP [20] and CCS [21], or temporal logics [22]. However, neither directly deals with the combination of services, hierarchical service composition, dynamic binding and policy-constraints via similarly simple mappings as we have established for stream-processing functions and their composition and refinement operators. The same is true for state-machine based service semantics as they are found, for instance, in [23], which do provide clear definitions of services and their assume/guarantee interfaces, but omit the notions of dynamic binding and scale that are crucial for addressing the full SOA context.

Recently, several approaches have been proposed to orchestrate services. One example is WS-BPEL [24] [25]. Another one is the Orc [26] [27] orchestration language. Compared to WS-BPEL, Orc is inspiring in its concise yet semantically deep expression of a broad range of workflow patterns. Other approaches, such as Live Sequence Charts [28] and Executable UML [29], support specifying interactions and workflows using graphical notations. The use of a graphical version of the interaction specification language for ORS is an area of future work.

None of the mentioned techniques has explicit support for service definition, composition, hierarchical decomposition, separation of concerns and dynamicity *within* a single semantic framework, let alone modeling language.

ORS introduces this combination and also provides an Architecture Definition Language as an embedded DSL within Clojure.

While this semantics and ORS language captures the essence of services and SOAs, there is plenty of room left for further refinement. We have left the channels within the system model untyped, for instance; explicit typing would allow us to further explicate the role of the Service/Data connector of a Rich Service (or, analogously, the function $r$ in the definition of a SOA). For instance, we could distinguish between channels that carry application data from those that carry messages controlling the system's topology.

The semantics currently rests on a simple model of causality, modeled via streams whose domain is the set of natural numbers, and whose range consists of finite message sequences. Clearly, this can be refined further into streams that reflect the needs of real-time systems. One such refinement would use the set of real numbers as the domain, and have also infinite message sequences as the range (again using the real numbers as the domain of these sequences).

**Rich Services and Aspects.** The Rich Services and ORS both support separation of concerns – infrastructure services are injected into the composition of application services. A similar capability is available in AspectJ [30]. While both ORS and AspectJ target Java virtual machines, ORS has an architecture view and

injects behavior at the service interaction level while AspectJ injects functionality at the Java method level.

**Services and feature composition.** The complex issue of composing services or features has been addressed in the seminal Distributed Feature Composition (DFC) [31] [32] and Feature Oriented Model Driven Development (FOMDD) [33]. ORS extends the functional feature composition approach of DFC and FOMDD by addressing both dynamic service binding and explicit treatment of crosscutting concerns.

**Scale** We introduced Rich Services as a hierarchical pattern for decomposing services while at the same time disentangling application and infrastructure concerns within the Rich Service. We showed that this decomposition maps directly to a service function decomposition into a superordinate and a subordinate function such that the superordinate function caters to the infrastructure concerns, while the subordinate function represents the application-level concerns. Exploiting function composition (with explicit interfaces via channel histories) is a powerful mechanism to express hierarchy.

The relationship between the super- and subordinate functions in describing SOAs in general, and Rich Services in particular, holds significant methodological potential. One interesting question is how to decide on the tradeoff between the responsibilities of the super- and the subordinate function in implementing the overall behavior of the SOA. A second one is how responsibilities of a superordinate function propagate through a hierarchically decomposed service.

Several frameworks and APIs have the ability to load, remove, and rewire services during runtime (such as OSGi [34] Java-based frameworks). In addition, ORS has the ability to deploy services across JVMs and comes with a dedicated DSL for modeling interaction patterns. Existing ESB implementations, such as Mule [35] and ServiceMix [36], come with DSLs to specify service interaction and deployment. ORS advances the state of the art by providing an interaction language. Furthermore, ORS can specify the topology of distributed systems in a single model. Finally, ORS comes with a Java based runtime system that can reuse connectors and components implemented in previous Java-based ESBs.

**Fit for Realistic Systems** The notion of Rich Services emerged from a number of systems engineering projects where we had to design and implement large-scale SOA integration projects [37]. There, Rich Services have proven its value in transparently decoupling application from infrastructure concerns across multiple layers of hierarchy, and have been key to quality assurance and principled evolution of the systems they were used on.

The semantics in this document was designed to allow us to capture the essence of systems built using Rich Services. We have yet to explore opportunities for *reasoning* about the resulting systems; however, the present semantics does lay the foundation for making headway in that direction.

**Relationship to MDD** Model-Driven Development (MDD) promises primarily lifting the granularity of discourse in designing and reasoning about complex systems to the level of a family of well-chosen abstractions (models) that

individually give a more tractable view of the overall system. In addition, MDD advocates the use of transformations from abstract to concrete models so as to automate as much as possible on the way from abstract specifications to concrete code.

In [38], we have shown how Message Sequence Charts (MSCs) and state machines can be mapped to a static version of the semantics presented here. Specifically, we have shown how to convert MSCs into state machines as a concrete model transformation example. This transformation could be expanded to the dynamic semantic model described here, and would thus directly capture the notions of creation and destruction of services at runtime. Clearly this would require changes to the syntax and semantics of both MSCs and state machines, but this does hold promise in reducing the amount of manual specification effort using the bare stream-based semantic model.

In general, it would be interesting to identify a set of description techniques that together concisely specify services and SOAs. It is not clear upfront what this set will include. Clearly, it will contain mechanisms for specifying functions, as well as function bindings – but what concrete shape these description techniques will take, and to what degree they will overlap is not immediately obvious.

**Layered Augmentation** We have chosen the semantics deliberately in a generic (one could argue: barren) form so as to avoid cluttering the notions of service and SOA with concerns that are not germane to the core of either.

With this generic service/SOA model in place we can now ask ourselves how we can augment it to cater to more refined service notions. This is easily possible. Consider, for instance, the three service/SOA notions introduced in Section 1.1. Our notion of service and SOA is compatible with each one of them. However, each adds terminology to capture a different aspect of what they consider essential: “business function”, “conceptual framework”, “abstract resource”, “provider entities”, “requestor entities”, “provider agent”, “ownership domains”, etc.

We argue that these terms can adequately layer on top of the core terms of services as functions, and SOAs as dynamic functional programs. As just one example, consider the term “business function”. Clearly, business functions can also be expressed as mathematical functions. The key difference to our generic semantic model is that it does not carry with it a *domain model* of entities and relationships that are of significance in the business domain within which the SOA is to be constructed. In this sense, the notion of business functions to define services emerges from our generic semantics for services and SOAs, augmented with a domain model that articulates the context within which the services and SOAs under consideration exist.

An interesting extension of the presented model is, therefore, one that embeds the given service/SOA notions within specific domain models.

## 7. Summary & Outlook

In this article, we have introduced a concise mathematical model for services and Service-Oriented Architectures (SOAs). Our premise was to map services to functions and SOAs to dynamic functional programs.

To that end, we have given a model for services as stream processing functions, and have shown how to model dynamic functional programs via a function binding environment for an initial pair of super- and subordinate stream processing functions.

We further exploited this semantics for dynamic functional programs to model hierarchical decomposition and separation of concerns as introduced in the context of Rich Services. This yielded a precise semantics definition for service integration at scale.

In addition, we showed how the presented service/SOA notion can be implemented using a domain specific language (DSL) for Rich Services, and have indicated how the syntactic and operational concepts of the DSL map to the semantic model – with "CellSense" as our running example.

There is plenty of opportunity for future work. Specifically, the relationship between super- and subordinate functions warrants further methodological treatment, with an eye towards a theory and calculus for policy specifications and enforcement in dynamic systems. Modeling notations for the transitions from one environment binding to another are also an interesting area for further research. Last, but not least, typing concepts for the channel valuations might lead to further structuring opportunities of service/data connectors in Rich Services.

## 8. Acknowledgments

The author is grateful to Manfred Broy, Barry Demchak, Bernd Finkbeiner, Bill Griswold, Markus Kaltenbach, Massimiliano Menarini and Bernhard Rumpe for stimulating discussions on this topic. This work was funded under the "PALMS" project funding from the Genes, Environment and Health Initiative NIH/NCI Grant U01 CA130771, under the "iDASH" project funding from NIH Grant U54 HL108460, under the "CitiSense" project from NSF Grant CNS-0932403, under the "MRI: Development of Instrumentation for Project GreenLight" project from NSF Grant CNS-0821155, under the "Foundations, Architectures, and Methodologies for Secure and Private Cyber-physical Vehicles" project from NSF Grant CNS-0963702, and by the California Institute for Telecommunications and Information Technology (Calit2).

## 9. References


1. IBM: WebSphere Process Server for Multiplatforms, Version 6.0.x. In: Introduction to WebSphere Process Server. Available at: http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.wsps.ovw.doc/doc/covw_intro_server.html

2. Web Services Glossary. In: W3C Working Group Note. (Accessed February 11, 2004) Available at: http://www.w3.org/TR/ws-gloss/

3. OASIS: OASIS Reference Model for Service Oriented Architecture 1.0. In: OASIS Standard. (Accessed October 12, 2006) Available at: http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html

4. Broy, M., Krüger, I., Meisinger, M.: A Formal Model of Service. ACM Transactions on Software Engineering and Methodology (TOSEM) 16(1), 5 (2007)

5. Broy, M., Stølen, K.: Specification and Development of Interactive Systems. Springer (2001)

6. Broy, M., Krüger, I.: Interaction Interfaces - Towards a scientific foundation of a methodological usage of Message Sequence Charts. In Staples, J., Hinchey, M. G., Liu, S., eds. : Formal Engineering Methods (ICFEM'98), pp.2-15 (1998)

7. Broy, M.: A Logical Basis for Modular Systems Engineering. In Broy, M., Steinbrüggen, R., eds. : Calculational System Design. IOS Press (1999) 101-130

8. Möller, B.: Algebraic Structures for Program Calculation. In Broy, M., Steinbrüggen, R., eds. : Calculational System Design. IOS Press (1999) 25-97

9. Stephens, R.: A Survey of Stream Processing. Acta Informatica 34(7), 491-541 (1997)

10. Rumpe, B.: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. TU München (1996) Dissertation (in German).

11. Arrott, M., Demchak, B., Ermagan, V., Farcas, C., Farcas, E., Krüger, I., Menarini, M.: Rich Services: The Integration Piece of the SOA Puzzle. In : Proceedings of the IEEE International Conference on Web Services (ICWS),

Washington, DC, pp.176-183 (2007)

12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)

13. Farcas, C., Farcas, E., Krüger, I.: Requirements for Service Composition in Ultra-Large Scale Software-Intensive Systems. In Choppy, C., Sokolsky, O., eds. : Foundations of Computer Software: Future Trends and Techniques for Development, vol. 6028, pp.93-115 (2010)

14. Demchak, B., Kerr, J., Raab, F., Patrick, K., Krüger, I.: PALMS: A Modern Coevolution of Community and Computing Using Policy Driven Development. In : 45th Hawaii International Conference on System Sciences (HICSS) (2012)

15. Farcas, C., Farcas, E., Krüger, I., Menarini, M.: Addressing the Integration Challenge for Avionics and Automotive Systems - From Components to Rich Services. The Proceedings of the IEEE Special Issue on Aerospace and Automotive Software 98(4) (April 2010)

16. Krüger, I., Demchak, B., Menarini, M.: Services for all! In Heisel, M., ed. : Essays Dedicated to Bernd Krämer on the Occasion of His 65th Birthday 7365. Springer, Lecture Notes in Computer Science (2012)

17. Hickey, R.: Clojure. In: Clojure Website. (Accessed 2010) Available at: http://clojure.org/

18. Halloway, S.: Programming Clojure. Pragmatic Bookshelf (2009)

19. Noël, C.: Extensible software transactional memory. In : Proceedings of the Third C* Conference on Computer Science and Software Engineering, Montréal, Quebec, Canada, pp.23-34 (2010)

20. Hoare, C. A. R.: Communicating Sequential Processes. Prentice Hall, New York (1985)

21. Milner, R.: Communication and Concurrency. Prentice Hall, New York (1989)

22. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer Verlag, New York (1992)

23. Can, A., Halle, S., Bultan, T.: Modular Verification of Asynchronous Service Interactions Using Behavioral Interfaces. IEEE Transactions on Services Computing PP(99) (2012)

24. OASIS Web Services Business Process Execution Lang: Web Services Business Process Execution Language Version 2.0. In: OASIS Standard. (Accessed April 11, 2007) Available at: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

25. Rosenberg, F., Dustdar, S.: Business Rules Integration in BPEL - A Service-Oriented Approach. In : Seventh IEEE International Conference on E-Commerce Technology, Munich, pp.476-479 (2005)

26. Kitchin, D., Quark, A., Cook, W., Misra, J.: The Orc Programming Language. In Lee, D., Lopes, A., Poetzsch-Heffter, A., eds. : Proceedings of FMOODS/FORTE 2009, Lisbon, Portugal, vol. LNCS 5522, pp.1-25 (2009)

27. Kitchin, D., Cook, W., Misra, J.: A Language for Task Orchestration and its Semantic Properties. In : Proceedings of Concur'06, Bonn, Germany, pp.477-491 (2006)

28. Harel, D., Marell, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Berlin, Germany (2003)

29. Mellor, S., Balcer, M.: Executable UML. Addison-Wesley Pearson Education, Indianapolis, IN, USA (2002)

30. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: Getting started with ASPECTJ. Communications of the ACM 44(10), 59-65 (October 2001)

31. Jackson, M., Zave, P.: Distributed feature composition: A virtual architecture for telecommunications services. IEEE Transactions on Software Engineering 24(10), 831-847 (October 1998)

32. Zave, P.: Modularity in Distributed Feature Composition. In : Software Requirements and Design: The Work of Michael Jackson. Good Friends Publishing Company, Chatham, New Jersey (2010) 267

33. Trujillo, S., Batory, D., Diaz, O.: Feature Oriented Model Driven Development: A Case Study for Portlets. In : Proceedings of the 29th International Conference on Software Engineering (ICSE2007), Minneapolis, MN, pp.44-53 (2007)

34. OSGi Alliance: OSGi Service Platform Core Specification. In: OSGi Alliance. (Accessed June 2009) Available at: http://www.osgi.org/download/r4v42/r4.core.pdf

35. mulesoft.org: Mule ESB. In: mulesoft.org. Available at: http://www.mulesoft.org/

36. apache.org: ServiceMix 4. In: apache.org servicemix site. Available at: http://servicemix.apache.org/

37. Demchak, B., Ermagan, V., Farcas, E., Huang, T.-J., Krüger, I., Menarini, M.: A Rich Services Approach to CoCoME. In Rausch, A., Reussner, R., Mirandola, R., Plášil, F., eds. : The Common Component Modeling Example, Comparing Software Component Models LNCS 5153. Springer (2008) 85-115

38. Krüger, I.: Distributed System Design with Message Sequence Charts. Technische Universität München (2000)