FOUNDATIONS OF COMPUTING AND DECISION SCIENCES
Vol. 40 (2015) No. 4
DE GRUYTER OPEN
DOI: 10.1515/fcds-2015-0015
ISSN 0867-6356
e-ISSN 2300-3405

# TOWARDS A PROCESS CALCULUS FOR REST: CURRENT STATE OF THE ART

Dariusz DWORNIKOWSKI Andrzej STROIŃSKI Jerzy BRZEZIŃSKI *

**Abstract.** SOA is a popular paradigm for building distributed systems that has gained a great recognition over past years. There are two main approaches to implementing SOA: SOAP-based and RESTful Web services. In order to address problems of modeling and verification of Web services, several process calculi have been proposed for SOAP-based Web services but none for the RESTful Web services based systems. This article is a comparative survey on existing process calculi for SOA systems, also the existing attempts to formalize REST systems are discussed. The aim of the article is to see how process calculi for SOAP-based systems deal with different aspects of their modeling domain, and whether their approaches can be used to model RESTful and ROA systems. Finally, basing on the survey, requirements for a new process calculus specific for REST are defined.

**Keywords:** REST, ROA, process calculi, SOA, formal modeling

## 1 Introduction

SOA is a popular paradigm for building distributed systems that has gained a great recognition during the past years. The basic unit of computation in SOA is an autonomous and independent service. Service provides its functionality through a clearly defined interface. Clients, such as other services or end users, access service interfaces using a communication medium, and a well known access protocol. Services in SOA are loosely-coupled, i.e. their functionality does not depend on other services. In addition, such services can be independently developed and maintained in the form of application components, called Web Services (WS). Next, such Web Services are deployed in application servers (like Apache Tomcat), and shared with other developers. As a result, in order to provide a more sophisticated functionality than a single

*Institute of Computing Science, Poznań University of Technology, Poznań, Poland, {dariusz.dwornikowski, andrzej.stroinski, jerzy.brzezinski}@cs.put.poznan.pl

service, services need to be composed into business processes, i.e. workflows where services act as clients to other services. Business processes can be either composed by means of orchestration or choreography.

Orchestration distinguishes one of the services to play the role of an orchestrator, which manages invocation of other services according to an execution plan provided by the programmer. Orchestrators are often called business process engines, and have quite many implementations: BPEL [3], Jolie [26], JOpera [27], and ROsWel [9].

On the other hand, in choreography services determine next steps of business process execution independently, basing only on their local knowledge. In consequence, such an approach can be more efficient because of the lack of business process engine, however the management and maintenance cost of such systems is much higher. This is why choreography is practically not used in production.

There are two main approaches of implementing SOA: SOAP-based and RESTful Web services. SOAP-based systems are highly standardized and use SOAP (*Simple Object Access Protocol*) as a communication medium. SOAP provides means to execute methods provided by a services' interfaces. In this approach, WSDL (*Web Services Description Language*) is used to describe service interface in XML format, giving a way to automatically access the knowledge of service's functionality. WS-BPEL engines are common orchestrators used to compose services into business processes in SOAP-based SOA systems. WS-BPEL language describes procedurally when and which services are execute during a business process execution. In this approach, HTTP protocol is used only as a transportation layer. SOAP protocol message semantic is encapsulated in the form of so called SOAP envelope (in the form of an XML document), and placed in the *DATA* field of HTTP message. Unfortunately, SOAP is only a communication protocol, so in order to provide more sophisticated mechanisms, a huge stack of standards for SOAP-WS has been defined, often addressed to as WS-*. In consequence, in the context of large scale Web applications, SOAP and WS-* are not efficient in terms of computational and communication costs.

In order to address the problem of complexity and difficulty in using SOAP-based systems Roy T. Fielding introduced REpresentational State Transfer (REST) [18]. REST does not use the whole stack of XML-based WS-* protocols, hence it is more "lightweight" and leads to easier and cheaper development, and maintenance. RESTful Web Services (Web Services implemented according to REST model) organize functionality into collection of invocable resources and relationships among them. A resource is the main element of abstraction, rather than a service. All resources are uniquely identified, and are hierarchically composed. In REST, the set of basic operations limited to CRUD (Create, Read, Update, Delete) operations, more complex operations are possible to implement by executing basic operations in chains (e.g. POST-GET to check your writes). The most common implementation of REST is ROA (*Resource Oriented Architecture*) which uses the existing Web infrastructure, specifically the HTTP protocol [38]. In ROA, CRUD operations are mapped onto HTTP methods, i.e. GET, POST, PUT, and DELETE, etc. In fact, ROA approach also adds guidelines how to use, HTTP methods other than CRUD but this is only introduced as neat pick to use RESTful systems in a more user-friendly way. Finally, the unique addressing and hierarchy property of RESTful systems is supported by

the URI standard. More on SOAP-based systems, REST and ROA, and differences between the two main approaches to SOA can be found in [35] and the Section 2 of this paper.

Both in SOAP-based and REST approaches however, the same problems with service composition can occur, and in the result can yield an incorrect system behavior. In the case of large scale systems some of the advantages like modular architecture, incremental and continuous development (introduced by SOA approach independently) lead to undesirable situations like cyclic invocations and/or missing service invocation. There is also a problem of automatic, or semi-automatic, process composition that fulfils certain requirements, such as safety, liveness, or widely understood correctness. These problems can be addressed by understanding how services behave, how they communicate within a business process, and finally what are the correct runs of business processes. Among various semi-formal techniques, and algorithms, number of formal methods, mainly process calculi, has been proposed to model and understand behavior of services and business processes. Process calculi can be used to model a system composed of services prior to its implementation to verify its properties with model checkers and specialized verification tools. These formal models can be also used to verify whether the system described in a specification is satisfied by the implementation. Finally, process calculi are a promising modelling paradigm to be used in a booming research areas of process mining, and in particular, service mining, where process models are discovered based on logs gathered from a system.

A number of process calculi has been proposed for modelling key aspects of SOAP-based SOA systems, in particular orchestration, coordination and conversations of Web Services. In this paper we specifically focus on the following formalisms that in fact are the only existing ones: the Conversation Calculus (CC) [45], SOCK [22], COWS [33], SCC [6], CaSPiS [7]. Unfortunately, these process calculi are not suitable to model ROA systems and RESTful Web services. We are however interested whether some parts of these calculi could provide an inspiration to create a process calculus specific for ROA and REST. Unfortunately, there has been little research on formalizing behavior of RESTful systems. The most closely related papers on the topic are [30] and [25]. Also, to some extent, formalization of RESTful systems is also present in work [34, 40, 31] but the focus there is put on describing RESTful APIs and enhancing it with semantic information.

The paper is structured as follows. Section 2 discusses differences between SOAP-based Web services, RESTful and ROA, in Section 3 we give a short introduction into what process calculus is. Section 4 briefly discusses the process calculi analysed in the paper. A framework for comparing and analysing the calculi has been presented in Section 5, whereas Section 6 contains the comparison of process calculi for SOAP-based systems, and Section 7 discusses approaches to formalize REST and ROA systems. Finally Section 8 contains concluding remarks.

## 2 Differences between SOAP-based and RESTful Web services

The great advantage of SOA compositionality. Service composition into business processes often has a formal basis that allows users and developers model distributed systems structure and behavior. Thanks to that, the formal verification and further enhancement of SOA system is possible. Unfortunately, SOAP-based and RESTful systems are significantly different, consequently preventing using methods specific for SOAP-based approach in REST and ROA, and possibly vice-versa. In the context of this paper, it is crucial to understand major differences that make ROA different from SOAP-based systems. Below we provide a list of these differences:

**Different granularity and first class citizens:** From the client point of view, in REST and ROA resources are first class citizens in the system. There is no notion of a service (as in SOAP), understood as a closed entity with a closed set of functionalities, and a clearly defined location. In REST, location (hosts) of resources and sub-resources is not known, even resources that are semantically close can be in fact places in completely different hosts. This has also another important consequence, in SOAP there is a one to one relation between the application process and its service, in REST on application process can be behind a bunch of resources.

**Hierarchical resource dependency:** In ROA, the whole functionality is provided in a form of resource collection available for invocation in some order by a client. REST and ROA uses URI to address resources, natural consequence of URI addressing is hierarchy. This causes some resource to be simply a sub-resource of others.Correctly modeled and implemented RESTful system will use URI addresses to pinpoint inclusion of some of resource inside other ones. In SOAP services are closed and non divisible entities, there is no hierarchy among them.

**Resource representation and HATEOAS:** In REST every resource can have many representations, which are produced to a client, depending on what is needed. If a client is a Web Browser used by a human, resource can show its HTML representation - a Web page as we know it. When accessed by a computer program, the same resource can produce an XML or JSON representation that is known to be easy to be processed by a computer. Which type of representation is showed is requested by a client, which sets the mime type of the requested resource representation typically in the *Accept:* header of the request [24]. Different resource representations may contain links to other resources, just as a Web page contains links to other Web pages. Client can then automatically follow the links and "uncover" its path in the processing of the business process it executes. This property is called *Hypertext as the Engine of Application State* - HATEOAS, and is very specific to ROA [38]. Thanks to that a client does not have to posses the whole knowledge of every possible step in a business proces but rather can learn it from other resources in forms of links.

**System logic on the client side:** The system execution knowledge is stored at client's side (calling resource), in contrary to SOAP-based services where the execution knowledge is stored in a server, where a particular SOA service is deployed. Thus, the business process logic is executed at the client side (active) due to the fact that client simply operates on resources by invoking them using a finite set of predefined operations. In consequence, a client stores information about the workflow execution

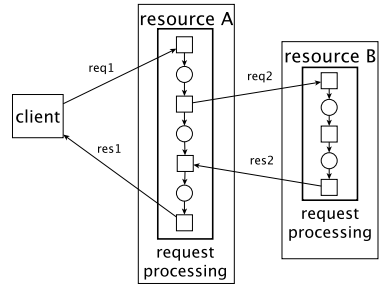



**Figure 1**: A resource acting as a client to another resource

between resources. It is a client's responsibility to pass the output data from one resource to the input of other resource, and to follow links from one resource to another, according to HATEOAS property. Effectively, a business process in ROA is a resource, and can be nested in another process resource, acting as a client towards it.

**Passive role of resources:** Resources in ROA are passive, they only provide appropriate representation and implementation of each available operation, and execute it only on client's demand. Such an approach introduces unique, and often desirable properties like stateless communication and unified interfaces. Detailed description of these properties and their importance is discussed in [18].

**Business process is a resource:** In order to achieve complex functionality in ROA, client composes resource invocations (consist of predefined set of operation CRUD) into workflows or business processes. In consequence, resources can act as clients and compose desired functionality using other resources. A client resource will further be called a *process resource*. In Figure 1, `client` invokes passive resource (`resource A`), which invokes resource (`resource B`) in order to fulfil request from the `client`. Finally when `resource B` sends the response, `resource A` is able to send the response to client's request.

**Process resources may be nested:** Resources and process resources may be nested, and be further orchestrated into more complex process resources. Here resources act as clients to other resources. In such a case, process logic is also nested in internal events of resources. A Petri Net in Figure 2 shows an example of how important local events are. First, there are three cooperating resources: `resource A`, `resource B` and `resource C`, where `resource A` plays a role of business process resource composing two other resources with logical `or` workflow pattern. The events `A::b` and `A::d` are in alternative path of execution of business process at `resource A`, thus only one of them will be executed. In consequence sending the message `req1` to `resource B` or message `req2` to `resource C` will occur depending on the decision made in state represented as place $p(\{A :: a\}, \{A :: b, A :: d\})$. In the alternative case in which there are additional events `A::a'` and `A::a''` the decision about which resource (`B` or `C`) to invoke later is done in a place $p(\{A :: a\}\{A :: a', A :: a''\})$.

**HTTP protocol as a communication layer:** ROA system is constructed from

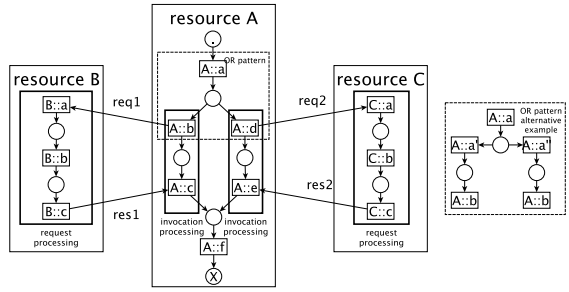
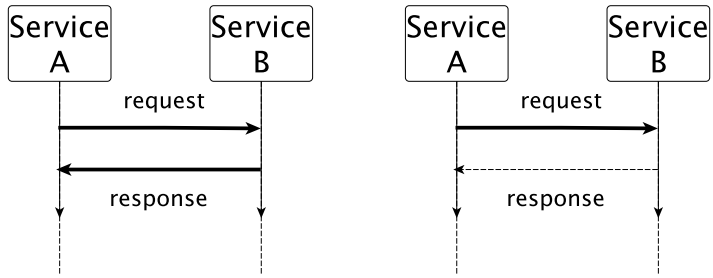



**Figure 2**: Nested business process, and local events dependencies



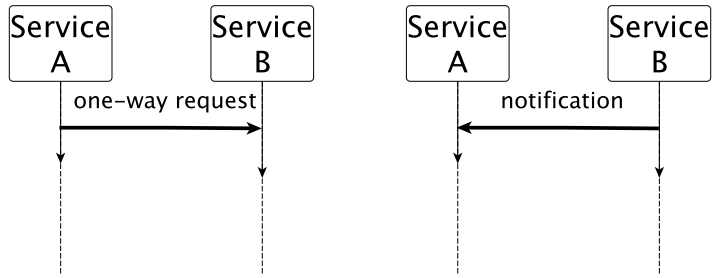

(a) sync. two-way interaction (b) async. two-way interaction

**Figure 3**: Two-way service interaction model [14]

a collection of interacting resources. During this interaction, resources use HTTP protocol as a communication medium. Semantics of HTTP protocol is used in order to determine how to handle the request [17]. Some HTTP methods are required to be safe, i.e. they do not modify the resource, which means they can be cached, prefetched without any changes in the resource. The only safe methods are GET, OPTIONS, and HEAD. There are also methods, which are idempotent, meaning that they can be called multiple times on a resource without different results, e.g. GET, DELETE, and POST are examples of idempotent methods.

Using HTTP as a protocol leads to a conclusion that formal model should be able to capture message semantics and make it possible to compose a simple set of messages into a more complex one with respect to commonly known process composition structures: OR, AND, XOR, loops, etc.

**Service interaction model:** According to [14] SOAP-WS defines two types of service interaction patterns: two-way and one-way. First one, is a simple communication where service A invokes service B and later service B sends a response to the service A. This case is depicted in Figure 3. This interaction pattern can be further divided into two sub-patterns: synchronous or asynchronous communication.




(a) request one-way interaction (b) notification one-way interaction

**Figure 4**: One-way service interaction model [14]

The second type of service interaction is one-way communication. Here the subtype of interaction depends on the service point of view. Lets consider Figure 4a. Here `service A` simply invokes `service B` and does not receive response. In the Figure 4b the role of service is reversed. The `service A` passively waits for invocation from `service B` in order to become notified. In consequence, if some service requests another service with one-way request invocation pattern, from the requested service point of view the invocation pattern used is notification.

In contrary to SOAP-WS interaction patterns, in ROA systems resources communicate using HTTP protocol, and its methods. In ROA, each time a resource is invoking another one with a HTTP method, it immediately needs to get a response; this is a synchronous communication pattern. This does not mean that RESTful Web Services cannot communicate asynchronously but this type of communication is stimulated by a series of synchronous communications (Figure 5).

In Figure 5, event `A::a` of sending message to `resource B` occurs in `resource A`. Next, in `resource B` message from `resource A` is received. Then, internal processing in `resource B` takes place (`B::b`). In addition, it is important that in `resource A`, during communication, any internal processing between events `A::a` and `A::b` is not possible because of request-response communication model of HTTP protocol. When internal processing in the `resource B` is finished it sends (`B::c`) back the response back. `Resource A` receives the result in event `A::b`. Further, there is a second analogous synchronous call to the same resource. Both call, together, can simulate an asynchronous call to a resource.

In practice these two synchronous calls would be implemented in HTTP, using its 202 return code. The first call would get a 202 return code (Accepted). This code means that a request has been accepted but not processed. Client would however get a link with an address where it should expect the answer. The second synchronous call would be to get the answer.

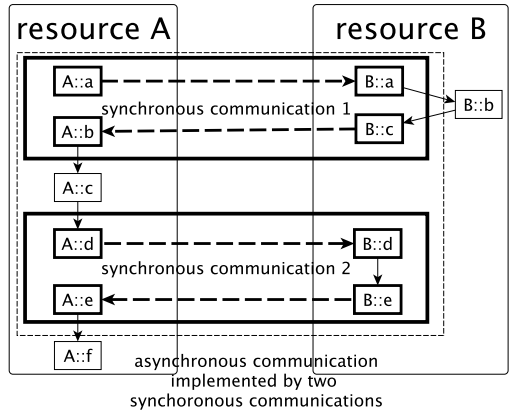



**Figure 5**: Asynchronous communication implemented by a sequence of two synchronous communications.

## 3 Introduction to Process Calculi

Process calculi are formal languages for algebraic representation and reasoning about processes and their behavior [19]. In process calculi processes are defined using symbolic terms, these terms are manipulated and composed using a set of operators. Operators, as well as terms provide a syntax of a process calculus, whereas semantics of the language is expressed with the use of structural operations semantics (SOS) rules [36, 19]. The rules define how a process calculi expression evaluates, and what is more important, how process behavior evolves. SOS rules induce a process graph (or a labelled transition system), which graphically represents a whole state space of a process with its all possible states.

Most of process calculi define a common set of operators, prefixing (.), alternative composition (+), and parallel composition (|). A process is defined by actions it can perform (ranged over $a, b, c..$). For example, a simple process that can perform action $a$ and $b$ in sequence, can be defined as $a.b$. Alternative composition, often represented by + adds non-determinism, so process which can perform either $a$ or $b$ is defined as $a + b$. An infinite behavior can be expressed using recurrence, e.g. $P = a.b.P$ states that process can perform actions $a.b.a.b.a.b..$ ad infinitum. Two processes that work in parallel can be written as $P \,|\, Q$. If we define $P = a.b$ and $Q = c.d$, possible transitions of $P \,|\, Q$ will be:

$$P \mid Q \xrightarrow{a} b \mid Q \quad P \mid Q \xrightarrow{c} P \mid d$$
$$b \mid Q \xrightarrow{b} Q \quad b \mid Q \xrightarrow{c} b \mid d \quad P \mid d \xrightarrow{a} b \mid d \quad P \mid d \xrightarrow{d} P$$
$$b \mid d \xrightarrow{b} d \quad b \mid d \xrightarrow{d} b \quad P \xrightarrow{a} b \quad Q \xrightarrow{c} d$$
$$b \xrightarrow{b} \surd \quad d \xrightarrow{d} \surd$$

What distinguishes process calculi from other formal modelling techniques is abstraction, compositionality, extensibility, and compactness.

When modeling a larger system, where many processes communicate, and synchronize to accomplish a common goal, it is sometimes hard to model every possible behavior. In fact, not every possible behavior is crucial from the point of view of a modeller. For example, in verification of a network protocol for inter-process communication, one is interested in actions in processes that conform to communication, i.e. sending, receiving, delivering and waiting for messages. Other actions, internal actions of processes, such as writing to disk, memory access, and logging may be insignificant from the point of view of the modelled protocol. Process calculi provide abstraction mechanisms that give a possibility to *hide* certain behaviors in a model thus simplifying it [4].

Compositionality in process calculi gives the possibility to create larger models consisting of independently created process models. Larger models are composed using composition operators for communication, synchronization or parallel composition. When combined with abstraction, compositionality becomes a powerful tool for refining and specification testing. One can seamlessly replace a single part of a system's model with more detailed one, without altering the rest of the model. Vice-versa, one can replace a detailed part of the model with a simplified one, using either abstraction mechanism of action hiding, or a new process modeled without unneeded behaviors [39].

Process calculi are extensible. New operators and syntactic elements can be added to facilitate new domain specific areas of modelling. Due to this feature there has been a plentiful number of extensions to the legacy process calculi that add new features, such as modelling of mobile processes, asynchronous communication, or modelling SOA systems.

Finally, process calculi are compact. Symbolic, textual representation is easier to process by computers, they take less space in memory, and even can easier to read by a human (e.g. in contrast to really big graphical Petri net models). Symbolic, textual representations are also ideal for automatic manipulation with the use of language processing algorithms and expressing infinite behavior using recursion operators.

## 4 Process calculi for SOAP-based systems and formalizations of REST

In this Section, approaches to formalize SOA systems implemented according to both SOAP and REST are introduced with short indication of the most important properties of each of them. Up to date, these are the only process calculi that are directly coping with the problem of formal modeling of SOA systems.

The Conversation Calculus (CC) proposed in [45] extends $\pi$-calculus to address central features of the SOAP-based systems. The novelty of the proposed solution is, according to the Authors, a new notion of conversation context in a SOA system. The central features of SOA, highlighted by CC, are distribution, process delegation, communication, context sensitiveness and loose coupling. Distribution is understood as an ability of a system to delegate certain activities to an external service provider, without engaging local resources. Authors notice that SOA's basic communicating mechanism is message passing but on top of it remote procedure call (RPC) and remote method invocation (RMI) are implemented. They focus on representing process delegation in their language, which is also a basic feature of $\pi$-calculus. Context sensitiveness is understood here as a space where computation and communication take place, context can be spatial but also behavioral.

Service Oriented Calculus Kernel (SOCK) [22] is a three layered calculus with formal semantics. Authors decompose SOA systems into three fundamental aspects: the behavior, the declaration and the composition. Each of these aspects can be designed and implemented independently of the other ones using SOCK. The three layers are represented by three calculi: *service behavior calculus*, *service engine calculus*, and *services system calculus*, are based on Author's previous work [10, 11, 21]. The first calculus deals with internal behavior of a service as well as communication primitives. Service engine calculus, on the other hand, is used for describing service deployment, i.e. parallel or sequential execution, state sharing, and sessions. Services systems calculus is used to define whole systems in terms of interactions between the services defined in *service engine calculus*.

Calculus for Orchestration of Web Services (COWS) [33] is a language which is strongly based and influenced by WS-BPEL, hence it is also suited to support only SOAP-based systems. In COWS, the basic elements are partners and operations but the basic computational entity is a service. The calculus supports shared states among service instances and allows one process to play more than one partner role. COWS borrows its constructs from well known process calculi, mostly $\pi$-calculus.

Service Centered Calculus (SCC) [6] is yet another calculus based on $\pi$-calculus , as well as Orc language [29]. SCC's Authors seek for a small set of primitives that make up a basis for formalising SOA systems. Their calculus is based on name passing and is able to explicitly express notions of service definition, invocation and bi-directional sessions. The novelty of the calculus is claimed to be the "*support for programming and composing services while taking into account their dynamic behaviour*".

CaSPiS (Calculus of Sessions and Pipelines)[7] is an evolution of SCC, it is also based on $\pi$-calculus, yet the Authors state two shortcomings of $\pi$-calculus when it comes to modeling SOA systems. SOA has "different first-class aspects" than $\pi$-

calculus, and $\pi$-calculus "communication primitives are too liberal", i.e. they lack structure of communication topology and hence they yield the analysis too complex. Therefore, CaSPiS's design is focused around three main SOA aspects determined by the Authors to be the most important. The first one is heterogeneity of services, i.e. each service has a full autonomy in proceeding with the computation or denying it (or request). CaSPiS tries to take the consequences of these decisions into account. The second aspect is client-service interaction. Here CaSPiS aims at supporting complex and safe client-service interactions, i.e. units of activity expressing conversations between a client and an instance of a service. The final aspect is orchestration, which is understood as an assembly of different services into a new one.

There has not been much research on formalization or process calculi for RESTful systems but one needs to mention two papers. In [30] Authors formalize RESTful application by proposing a formal model and temporal logic definitions of basic ROA (Resource Oriented Architecture) properties such as HATEOAS and statelessness. The purpose of the paper is to provide formalisation to support automatic verification of RESTful application behaviour, yet the paper does not define any semantics or syntax for describing the behaviour. The contribution is however valuable because of the proposed model that can be used to build a process calculi on top of it.

In [25], on the other hand, Authors present a formal model of semantic RESTful Web services equipped with syntax and semantics. However, they do not call this formalism a process calculus. Their model is based on triple spaces mixed with $\pi$-calculus and allows for "*complete and rigorous*" description of resource based Web systems. It is also a good basis to start a research on a formal process calculus for RESTful-based systems.

## 5 SOA calculi comparison framework

Every of the mentioned process calculi is different and tries to target different aspects of SOA. However, by looking closer, one can extract features that are common to all of the formalisms. We would like to introduce a comparison framework, where every feature that we compare has an obvious advocacy. Here we introduce several important aspects included in different process calculi which will be used for later comparison. For each of the criterion, a clear motivation of its importance in context of SOA and REST approach is presented.

**Communication** Probably one of the most important features of every process calculi for SOA is how it deals with communication. Although both SOAP-based and RESTful Web services use message passing as its basic communication model, more sophisticated communication schemes can be observed, such a one way communication in SOAP. The existing process calculi should offer a possibility to model these high level communication schemes, if not explicitly, then implicitly. Also semantic constrains of different methods should be taken under consideration. This is crucial from the point of view of their applications, i.e. discovering process models in service mining, verification of business processes,

liveness checking.

**Composition and hierarchy modelling** Business processes is a key aspect in SOA systems, they allow to compose a complex functionality using loosely coupled Web services. Composition can be done by means of orchestration, where a central element controls the flow of a business process, i.e. orchestration engine or a client. Processes can also be composed using choreography, where services determine by their own the next step in a business process. We analyze how orchestration and choreography can be expressed in the discussed process calculi.

**Equivalences and formal correctness** One of the key strengths of formal process calculi is a formal definition of equivalences on processes. We check which process calculi define equivalences, how strong they are and what they can be used for.

**Goals and domain specificness** All of the process calculi claim to address SOA systems. Some of them refine their domain of application strictly to SOAP-based systems. This is either stated directly in papers or can be deduced from various hints (such as WS-BPEL inspirations). We would like to see how much the existing process calculi are domain specific, and whether they are suitable to model other SOA approaches, such as REST.

**Tool support** Tools are GUI applications, modeling frameworks, model checkers supporting a given language, and libraries. In order to apply a formal language in practice, one needs a tool support. Tools can be specifically dedicated for a given process calculus, or an existing tool can be used. We check how tool support is fulfilled for the discussed process calculi.

## 6 SOA process calculi comparison

### 6.1 Communication

As far as communication is concerned we need to distinguish between communication understood in terms of process calculi as a flow of information between two (or more) processes, and communication among services. The first type of communication (we exploit name process communication) is a language element and one of the means to model interactions (along with synchronization). The second type of communication is a system specific feature that one wants to express in her or his models. Process communication can be used to model communication but it does not have to be the case. We analyze how different types/modes of communication in SOA can be expressed or whether they cannot be expressed at all. It does not matter whether a given calculus defines, or how defines, process communication operators, unless they limit or increase an ability to express communication directly. We are interested

**Figure 6**: Exemplary communication in SOCK

$$
\begin{aligned}
Client &::= getServerAddress@LB(\{\}, ServerAddress); \\
&\quad\ getData@ServerAddress(dataID, data) \\
LB &::= getServerAddress(\{\}, ServerAddress) \\
SomeServer &::= getData(dataID, data)
\end{aligned}
$$

in two aspects of communication, explicit communication modes, and possibility to model asynchrony and synchrony in communication.

In REST message semantics is important and has an explicit impact on behavior. For example GET method invoked on a resource will yield a completely different behavior of the resource than POST or PUT would. It is one of the most distinctive features of RESTful Web services when compared to those SOAP-based, where semantics of a service are in fact determined by its callable interface, e.g. method *add()* in a Calculator service will trigger a different behavior than method *divide()*. In SOAP messages carry data only, whereas in REST semantics as well. We check the discussed SOA calculi for supporting message semantics. Since they all are focused on SOAP-based systems only, they do not support this feature, because in SOAP-based Web services, which are procedural, this is not needed. Some of the calculi could be theoretically altered to accept a limited set of CRUD operations but their operational semantics would have to be reworked.

**SOCK** is the only one calculi which distinguishes explicitly different communication modes, yet they are based completely on those found in SOAP-based systems (especially WS-BPEL based). SOCK defines two modes of communication, one way and two way. Two modes are defined for both input and output, i.e. input operations supply a service functionality, and output operations are used for requesting service functionality. They are as follows (complementary for input and output): *One-Way/Notification* for one way mode, and *Request-Response/Solicit-Response* for two way communication. Such a duality in proposing complementary operations for input and output results from a channel like process communication. This seems a reasonable angle to determine direction of communication in SOA. Process communication in SOCK is synchronous but an asynchronous extension exists, proposed in [21]. An example of communication in SOCK, in Fig.6 can be found in [8]. We simplified the example to our needs. The example presents a simple load balancer scenario, where client asks a load balancer server for an address of a next (presumably less busy) server. After receiving a new address, the client can send a request with dataID argument, and get the corresponding data in return. *Client* is a process which sends a *getServerAddress* request to *LB* server. *LB* is a process, which accepts *getServerAddress* message and responds with a server address for a less busy server at the moment. *SomeServer* is a process which accepts *getData* messages.

**CaSPiS and SCC** take a more low level approach to communication, their aim is to provide a small, formal basis for reasoning about service oriented systems. Their

Authors do not distinguish among different communication schemes, the focus is put on expressing bi-directional sessions and different workflow patterns, as defined in [2]. The main focus in CaSPiS is put on communicating sessions using a $P > Q$ operator, which says that value emitted by $P$ can be consumed by $Q$. Another process communication is session synchronization, expressed by a channel like dual name operator but results in creation of a new session (or context). Taking this under consideration, both calculi seems to be way more universal than SOCK, and probably could be to some extent be used to model RESTful Web services by limiting the number of operation to the CRUD set used in REST. On the other hand, this would allow only for RESTful workflow modeling, without taking its distinguishing and features under consideration.

**COWS** takes its inspiration completely from BPEL, in fact, one can think of this process calculi as of a formalization of WS-BPEL. COWS tries to borrow from other calculi, such as $\pi$-calculus, update calculus, StAC_i, and $L_\pi$, and combine them to create a calculus to express BPEL orchestration. As far as communication is concerned, COWS support asynchronous and polyadic process communication. The modes can be executed by a service (invocation) or on a service (invocation of provided operations). Communication activities in COWS are invoked with communication endpoints, which are pairs $p \cdot o$, where $p$ is partner name, and $o$ is a operation. This is a strictly procedural approach, inherited from WS-BPEL.

**Conversation Calculus (CC)** is yet another calculus based on $\pi$-calculus. The basic mode of communication is message passing, as it was in all the previously discussed calculi. However, the Authors distinguish a higher level communication mechanism built on the top of message passing, which is service invocation. They see it as a "*delegation of an invocation of a whole activity, or process to a remote party*". The novelty of CC is that it strongly ties the communication and contexts, i.e. communicating inside context, and outside contexts. It distinguished between messages sent to and from contexts, so the direction is here preserved. From this perspective, CC can be a very interesting form the point of view of expressing models of RESTful Web services process instances discovered by means of process mining. Unfortunately, CC is another example of a calculus which does not provide explicit notion of communication schemes.

**Summary:** As far as modeling asynchronous and synchronous communication, none of the process calculi make an explicit distinction in its semantics. However, using basic process communication mechanisms, one could model synchronous and asynchronous communication "by hand", or using a simple pattern. An example in Fig. 7 shows two processes, for clarity we assume there is a shared action communication, i.e. processes synchronize on the same action name. Processes $Client_{sync}$ and $Server$ synchronize on *send* and *recv*, 0 is a null action, or termination. This pattern is a synchronous call, $Client_{sync}$ synchronizes with a $Server$ on send. $Server$ can now do action *process*, whereas $Client_{sync}$ needs to wait for response — *recv* action. In the second case, $Client_{async}$ makes an asynchronous call, between sending and receiving a response, it can do some action. This pattern can be used to model these two types of communication but without an orthogonal assumption that *recv* is a receive action for the *send* action, it would be not possible to connect these two

**Figure 7**: Synchronous and asynchronous example

$$Client_{sync} = send.recv.0 \quad Client_{async} = send.do.recv.0$$
$$Server = send.process.recv.0$$

events. For many cases it is sufficient to assume that orthogonally but there are scenarios, where this knowledge should be supported by language semantics. In service mining, a process mining sub field, it is one of key problems to match send and receive events in a log, and discover a process model from a log file containing such data. If there was a language supporting this, it would be possible to express asynchronous send/recv events, as automatically discovered from a log, more can be found in [1].

None of the discussed process calculi are really suitable for modeling communication in ROA systems, where messages have semantics. Modeling ROA would result in loosing a lot of information, due to strictly procedural treatment of messages in SOCK, CC, CaSPiS and SCC. Extending the existing SOA calculi could be possible yes to some extent pointless because of their basic assumptions about the modeled systems, where the message type is not important but rather the endpoint to which the message is directed. In ROA both these aspects are important but message type is crucial and results in side effects that cannot be neglected. For example, GET has no side effects on a called endpoint as it is only a "read" type operation, so it can only change the state of the caller. PUT or POST on the other hand change the state of the called endpoint but not necessarily the callee's state.

**Table 1**: Communication tabular summary

| Feat. | CC | SOCK | CaSPiS | SCC | COWS |
|---|---|---|---|---|---|
| Operation semantics | X | X | X | X | X |
| Type of comm. | async. | sync./async. | sync. | async. | async. |
| Explicit comm. modes | X | ✓ | X | X | X |

## 6.2 Composition and hierarchy modelling

Composition is a technique of constructing a business process from a set of loosely coupled services. It can be achieved either by orchestration or choreography. We analyze the existing SOA process calculi in order to see how services can be composed, is orchestration and choreography provided. We are also interested how legacy process calculi mechanisms are exploited to compose systems of services, whether a given process calculus uses typical composition operators, define its own, or the composition is more or less implicitly defined, i.e. it results from a specific semantics.

**CC** supports orchestration, and it is an example of a calculus where the notion of composition is implicit. CC provides neither traditional composition operators, used

in traditional process calculi, nor it defines its own. Instead, orchestration of services into more complex entities is carried out by using CC's communication operators, specifically $\leftarrow$ and $\uparrow$. Respectively, they denote communication with a dual, remote endpoint of a context, and communication with the enclosing context. The topology of a business process modeled in CC can be easily extracted following $\leftarrow$ operators in both modes (in and out). In order to extract the hierarchy of services, one can track $\uparrow$ operators. An implicit encoding of orchestration is quite elegant and provides a convenient and compact way for a modeler. On the other hand, in complex models it would be hard to instantaneously "spot" how a business process is composed, one would need to find the "entry" service, probably the one started with a client code, and track all the following communications to extract the hierarchy and topology of a process. Another disadvantage would be the flexibility in changing the process topology. The expressiveness of process calculi, where process composition is expressed explicitly, gives a possibility to rearrange how processes communicate and how they are related. In CC one would need to redefine the whole model to achieve that. For an extensive example of orchestrating services in CC see [45, 44], where an encoding of a flight booking BPEL process from [28] has been presented.

**SOCK** has an explicit notion of orchestration. Its *service system calculus* is specifically defined to address composition of different service engines defined in *service engine calculus* into a system. *Service system calculus* is based closely on *service engine calculus* and provides notion of executing a service engine (a service execution in a context) in a given location or in parallel to other service engine. For this a parallel operator || is used. How the services communicate and synchronize within the parallel composition is defined by operational rules specific for *service engine calculus*, i.e. concurrent and sequential with or without a persistent state. In summary, SOCK provides only means of orchestrating services, however choreography is (was) intended to be added by the Authors. Their previous work on $CL_p$ (*Choreography Language*) could be used to extend SOCK to provide means of expressing choreography, however choreography is not used practically in real life scenarios. The approach taken by SOCK has a nice feature that the system composition is defined externally to modeling of service behavior and contexts. However, the part of SOCK responsible for it is very limited, provides only parallel composition. Unfortunately, without analyzing service engine calculus models, one is not possible to determine the real topology of a business process and a flow of control among services. An example of using SOCK to model a system with orchestrating has been provided in [8].

**CaSPiS** similarly to CC has an implicit notion of orchestration, i.e. composition of services in rather modeled through communication than a system equation where services are composed. To model a flow of control, or composition, pipeline operators are used in CaSPiS, $>$ to synchronize one context with another. $P > Q$ will create a new instance of $Q$ whenever $P$ emits a new value that can be consumed by $Q$. Pipeline's semantics state that a $Q$ is replicated, so after consuming a value, $Q$ is still able to consume other values. Similarly to CC, the topology and control flow of a process composed with pipes can be extracted by tracking $>$ operators. Hierarchy on the other hand is achieved in CaSPiS by nesting contexts. The implicit notion of orchestration in CaSPiS results in the same problems as can be seen for CC, i.e. when

dealing with complex models it would be hard to immediately see the topology, also changes in topology could result in rewriting the whole model.

**SCC** which is a predecessor of CaSPiS does not have the pipeline operator, instead there is a service invocation operator $\Leftarrow$, which can be used to orchestrate services, along with a parallel composition $|$. Contexts on the other hand are used almost in the same manner as in CaSPiS, so the same problem of extracting control flow can occur.

**COWS** is again a language without an implicit notion of orchestration, instead invocation is used to build the flow of the composed process. This yields COWS models to be even harder to decode how services are orchestrated because of its lack of explicit notion of contexts and sessions. Due to this limitation COWS also does not support nesting services, so a hierarchy of processes cannot be encoded there. On the other hand, being a very basic calculus, COWS provides a standard library of patterns, which show how higher level constructs can be encoded in COWS, a full standard library can be found in [32].

Embedded processes is a feature which allows to express embedding of one process or service in another. This feature is important when one wants to model complex orchestration scenarios, where the topology of a business process is not "flat", i.e. where sub processes exist. Subprocess can be understood by a business process executed as a result of execution of a service. Additionally embedded processes are important from the point of view of modeling local, inner service behavior, where a service can be in fact composed of many application processes that compose its functionality. SOCK and COWS do not have explicit language elements to provide modeling of embedded processes, this information needs to be derived indirectly or added in terms of modeling convention. CaSPiS, SCC and CC allow to model nested processes, this mostly results from their direct focus on sessions and context, where nesting processes is a natural consequence.

**Summary:** All the discussed process calculi provide interesting ways of expressing composition, either orchestration or choreography. As it could be seen, there are two approaches, explicit and implicit one. Explicit is when the topology of composition is defined using separate language constructs. The implicit one is when the topology is defined by language operators used to model services. Both of the approaches offer interesting inspirations, the explicit one is more flexible when it comes to automatic model processing and model reusing. On the other hand the implicit approach seems more elegant and it would be easier to generate program code from such models. In the context of modelling ROA systems, both approaches are equally suitable, and the decision which one is better would be up to the modeler.

**Table 2**: Composition and hierarchy tabular summary

| Feat. | CC | SOCK | CaSPiS | SCC | COWS |
|---|---|---|---|---|---|
| Orchestration | ✓ | ✓ | ✓ | ✓ | ✓ |
| Choreography | X | ✓(possible with CL) | X | X | X |
| Embedded processes | ✓ | X | ✓ | ✓ | X |

### 6.3 Equivalences and formal correctness

Equivalences allow to determine whether two syntactically different processes (or in our case services) are equal, and in consequence can can can be substituted with one another. It is clearly an useful way to reason about SOA processes, reduce systems models, and provide modularity. Bisimulation is the best known and used type of equivalence relation in process theory field, it can compare two processes even if their branching structure is quite different.

**SOCK** does define directly any equivalence relations, yet in their previous work on $CL_p$ and $OL$ languages, SOCK's Authors defined the conformance relation, used to relate models of systems composed with orchestration and choreography [12]. The relation, called *comformability bisimualation*, is based on bisimulation (in fact it resembles branching bisimulation of [43]), and allows to determine whether an orchestrated system is equal, up to the names, and in terms of interactions, with the one composed by choreography. In order to relate one type of system to another, a *joining function* has been proposed, associating orchestrators from $OL$ with choreography roles from $CL_p$. The function is then used in the comformability bisimulation to test equality between the systems. The relation is strictly suited to check comformability but in fact it can also be used, to test equality between orchestrated systems. This can be done by neglecting the joining function or redefining it. In the result the bisimulation relation can be extended and used to compare plain SOCK processes.

**CC** defines equivalence directly in terms on strong and weak bisimulation. Additionally, extensive proofs that the strong bisimilarity is a congruence for all CC operators can be found in [44]. Based on that Authors define 7 process equations (equality laws), which further are used to prove some interesting spatial properties of CC, such as the fact that any CC processes are behaviorally equivalent to a process where the maximum nesting of contexts is two.

**COWS, SCC, and CaSPiS** do not define any equivalences directly but reasoning from the fact that they are based on $\pi$-calculus, some kind of bisimilarity notion for this process calculus can be used.

**Summary:** Relations that can be used to compare models or determine which models are equal, are very important from the perspective of model reduction and optimization. One can define reduction laws, and provide algorithms to minimize models, at the same time preserving the expected behavior. From the perspective of ROA modeling, standard bisimulation relation may not be fully suitable to compare ROA, and would need to be extended to catch semantics of REST methods, such as GET. GET method changes the state of the caller, not the callee, hence a service that receives one GET request intuitively should be behaviorally equal to the one that receives several GETs (to the same URI). This example shows that for ROA systems, behavioral equivalences and relations should be extended and redefined.

**Table 3**: Equivalences tabular summary

| Caclulus | Equivalence |
|---|---|
| CC | strong/weak bisimulation |
| SOCK | comformability bisimulation (through CL) |
| CaSPiS | X |
| SCC | X |
| COWS | X |

## 6.4 Goals and domain specificness

All the calculi have the same goal, modeling SOAP-based SOA systems, however they differ in syntax, semantics, and the assumed underlaying model. These differences largely result from assumptions and goals that certain calculus focuses on. Goals also determine how domain specific given language is, and what comes after that, how flexible it would be to use it outside of its base scope of application. From the perspective of modeling REST systems, it would be interesting to see which existing process calculi can step outside their application scope, and be useful to model some aspects of RESTful and ROA systems.

What all calculi have in common is that they all focus only on SOAP-based Web Services, i.e. procedural services. Some of them are however specialized in different aspects.

**CC** as the name suggests focused on conversations. Their Authors distinguish some key aspects of SOA systems and build the calculus around them. These aspects are: distribution, process delegation, contexts, loose coupling and communication. The main goal is to isolate these characteristics and reduce the expressiveness of CC to them in general. CC's motivation states that its main contribution is its notion of a conversation context, which can be seen as a medium where related interaction can take place. Interactions in CC are expressed as simple name passing, the impact in communication was put on communication between contexts, both inside and outside them. Summing up, CC can be seen as moderately domain specific, it assumes SOAP-based systems but gives general mechanisms that can be used to model other systems too. These assumptions however render CC to be not suitable to fully express RESTful systems, because of lack of the semantics of operations and pure procedural treatments of services.

**COWS** is another example of moderately domain specific process calculus. Although it is motivated to be strongly influenced by WS-BPEL, which supports only SOAP-based Web Services, COWS tries to provide "a foundation model, not specifically tight to web services' current technology". Similarly to WS-BPEL, COWS supports shared states among service instances, allows same process to play different roles, and permits programming stateful sessions by correlating different service interactions. As in the case of CC, modeling RESTful Web services in COWS is possible but with a major loss of interesting characteristics.

**SOCK** is composed of three calculi, each treating a completely different level of abstraction. Service behavior calculus is the most domain specific, as it inherits WS-BPEL specific communication primitives, such as one-way communication mode, although it offers also a request response model, which is the basic one in REST. Using it, one can easily model a RESTful Web service but again, some its characteristics will be lost. Second, and in third layer of SOCK, service engine calculus and service system calculus, respectively, are less domain specific. The first calculus expresses how sessions among services are correlated, whereas the latter one deals with composition of the whole system. These calculi can be seen as quite universal, and could be easily used to model systems outside SOAP-based scope, given the service behavior calculus is changed.

**CaSPiS** focuses on two aspects of SOA, modeling complex and safe interactions among services, and modeling orchestration in terms of data flow among different activities. To do this CaSPiS exploits sessions and pipelines, which are inspired by $\pi$-calculus and Orc [29] respectively. This general mechanism of modeling interactions and sessions is quite general, as inherited from SCC, which aimed at providing a small set of primitives that might serve as a basis for formalizing SOA systems. In this context, both CaSPiS and SCC are very universal, so they also allow to model systems outside SOA's scope. As far as modeling REST is concerned, as in the case of previously analyzed calculi, only the control flow and basic synchronizations can be modeled.

**Summary:** Summing up, CC, SOCK, CaSPiS (and SCC), and COWS are quite general, yet they are only suitable for modeling procedural systems, where services are defined either in terms of entities providing some callable endpoints, or in terms of "low" level processes, which synchronize on basic activities. None of the calculi make a distinction on semantics of operations, what can be seen as one of the distinctive features of REST, and in fact define the whole interaction flow.

**Table 4**: Goals tabular summary

| Calculus | Feature focus |
|---|---|
| CC | conversation contexts |
| SOCK | composition |
| CaSPiS | complex interactions |
| SCC | small footprint |
| COWS | WS-BPEL conformance |

## 6.5 Tool support

**JCaSPiS** is an implementation of CaSPiS process calculus as a JAVA-based framework, and can be used to implement SOA applications directly [5]. The usage of JAVA language results in portability across various platforms, and could be used in lead-

ing enterprise JAVA-based technologies. Furthermore, algebraic operators like parallel composition or restriction are already available out-of-box in the form of JAVA threads , as well as the `new` operator for instantiation of a new service instance. The additional implementation has been done to support inter alia `non deterministic choice` and `replication`. Next, the Authors consider connection protocols between hosts: TCP and HTTP. byte_code, XML and SOAP are supported as a service interaction protocols. Description on how main notations (like `Process`, `Service`, `session Context`, `PipeLine` etc.) of CaSPiS should be implemented in JAVA is discussed in detail in [5], there an example of implementation of a service calculating `sum` and `div` accordingly. In the context of SCC and CC Authors of JCaSPiS show how JCaSPiS can be easily adopted as a stand alone implementation of SCC.

In addition, it is worth noting that the SLMC model checker (*Spatial Logic Model Checker for Concurrency, Distribution and Mobility*) [13] developed for automatic verification of behavioral and spatial properties of distributed concurrent systems expressed in the $\pi$-calculus supports the specification of SOA systems in CC since version 2.01.

**JOLIE** is a service composition language based on [22, 10, 12, 21]. It combines achievements of the mentioned process calculi for SOA systems and introduces new language for Web service orchestration. In contrary to other similar languages, like BPEL or XLANG, its syntax is not XML-based but is C/JAVA-like what makes this approach easier to read and use by developers without a specialized graphical composition tool. In addition, it is important to note that strong formal background of this language allows to conclude and verify systems properties in automatic or semi-automatic way. JOLIE supports both types of communication, one-way and two-way, with so called `socked-based` communication primitives. The program is composed by `locations` - host address etc., `operations` - input and output, typeless `variables`, parallel process synchronization using `links` and `process definition`. The main statements according to process calculi that are implemented by JOILE are `sequence`, `parallel`, `non-deterministic choice`, and `synchronization` operators, for both in and out directions. JOLIE has an ability to integrate with JAVA programs what makes this approach very flexible and well suited for practical usage. Finally, full user guide for developers and users can be found on JOLIE homepage [26].

**COWS** has been provided with a CMC (*COWS interpreter and model checker*) tool that supports specification and verification of COWS models. The main functionality of this tool is to provide model checker of SocL formulas and to derive all computations originating from a COWS term in an automated way. Unfortunately this tools is a prototype stage of development and provides limited functionality.

**Summary:** In our opinion tool support is a very important aspect of every process calculus. Having a usable tool is needed to provide the community a chance to choose the given formalism and verify it against others but also give the possibility to practically evaluate the formalism on real life problems. From this perspective SOCK seems to be standing out by providing tool, which can be used for orchestration. From the perspective of modeling ROA systems, neither of the tools is suitable.

**Table 5**: Tools tabular summary

| Feat. | CC | SOCK | CaSPiS | SCC | COWS |
|---|---|---|---|---|---|
| Tool | SLMC | Jolie | JCaSPiS | JCaSPiS | CMC model checker |

## 7 Current approaches to formalize REST and ROA

Unfortunately, there are no process calculi for REST, therefore it was not possible to include them in Section 5. Instead, we will discuss some related work in the area of formalizing parts of RESTful behavior and some of REST and ROA properties in general. The research selected here, is often related to other areas than modelling SOA systems or verifying its properties, yet sometimes introduces some interesting aspects that should be included in formal description of ROA / RESTful system.

In [30] a formalization of RESTful behavior has been presented. The aim for the Authors was to formalize REST and ROA behavior in order to provide an automated verification of a behavior of a RESTful service. The article introduces a formal model of a RESTful interaction, called *resource-based application*, which is a tuple $RS =< R, I, B, \eta, C, D, \sim, OPS, RETS >$, where $R$ and $I$ are set of resources and their identifiers respectively, $B$ is a set of root identifiers, $\eta$ is a naming function mapping $I \to R$, $C$ is a set of client identifiers, $D$ is a set of data values, and $\sim \in (D \times D)$ is a equivalence relation on data values. Finally, $OPS$ and $RETS$ are finite sets of methods and return codes. Mind that this model is universal for all resource applications, it does not strictly specify RESTful systems, by doing that it follows a direction stated in [18] that REST is independent on the implementation. To extend $RS$ to be more RESTful, $\sim$ is extended to include resource representations. Resource representation in $RS$ is denoted by tuples $< id, d >$, where resource id is bound to the data. In order to be able say that two resource representations are in fact equal, $\sim$ is extended, as follows: $< id_1, d_1 > \sim < id_2, d_2 > \iff id_1 = id_2$ and $d_1 \sim d_2$.

Authors define communication as client-server, represented by a request response mode: $c :: op(i, args)/rc(rvals)$, where $c \in C$ is a client identifier, $op \in OPS$ is an operation (or method), $i \in I$ is a target resource identifier, $args$ are arguments and $rc \in RETS$ is a return code, $rvals$ are return values in a form of a list. Resource identifiers are associated with every communication as follows: $L(m)$ and $UL(m)$ denote linked and unlinked resources. Linked resources are those that are known to a client, and include resources ids returned as a result of a given communication. Unlinked resources are revoked at the client.

The $RS$ model is very general and can be seen as a template for all resource based systems. By parametrizing it and making concrete assumptions one can "generate" a model specific for REST or ROA. For example, by limiting $OPS$ to HTTP methods, and providing a mapping function to allow certain codes for certain methods, one can have a model for REST over HTTP. Additional parametrization can be done for HATEOAS property, for which linked and unlinked resources provide a good formal basis. Authors of the article do not specify any behavioral semantics for the model,

they only assume that the communication can be a CSP style synchronization. They define however what is a *RESTful behavior* in terms of temporal logics semantics, using CTL [16] and LTL [37]. In CTL they define certain properties of RESTful behavior: stateless behavior, hypertext-driven behavior, safety and idempotence, and a naming independence, as a consequence of the above. The rest of the paper concerns automated verification of a RESTful application, which is not a concern for us.

Although, the paper does not define a process calculi or operational semantics for modelling ROA systems, it is very interesting, as it defines a formal model that can be used as a basis to built on. The approach taken by the Authors is, in our opinion, very accurate. They provide a formal model for a very general resource based application, and by parametrization of the model one can generate a whole family of models describing various systems, such as REST of even ROA. It would be a good starting point to create a process calculus for REST and ROA systems.

Another article worth mentioning is [25], where Authors formalize RESTful Semantic Web Services, a RESTful Web Services flavour where resources are described with a semantic information provided in RDF or any other ontology definition language. In contrary to the previously discussed article, this one define two process calculi, one based on triple spaces paradigm, based on tuple spaces paradigm [20], and the second one based on $\pi$-calculus, which extends the triple space calculus with named channels and mobility characteristic for $\pi$-calculus. Both calculi are valuable lesson, especially using triple spaces merged with $\pi$-calculusto express creation and deletion of resources with POST and DELETE methods. However, the Authors' focus seems to be more targeted at providing semantic information than to formalize RESTful systems fully. They assume a HTTP medium for their model, which is in fact characteristic for ROA, yet they do not approach the problem of statelessness of resources, idempotence of some methods, and HATEOAS property. In the result, when not considering the semantic description added by triple spaces, the calculus is as expressive as plain $\pi$-calculus. On the other hand, it is a good inspiration to create a full process calculus for resource based systems, ROA in particular.

In [31] Authors propose a solution to describe RESTful APIs in a machine readable format, using hRESTS microformat embedded directly in HTML documents. The motivation for the paper was to provide a tool for API producers to publish their API descriptions in a format that could be easily consumed by clients in a an automatic way. The paper defines a simple semi-formal model of RESTful Web service using RDFS (*Resource Desription Framework Schema*). The model defines a RESTful Web service as an entity that publishes methods, which can be invoked using certain HTTP operations. This rather static approach to defining Web services is sufficient for APIs, it is however not sufficient to inspire a full model of RESTful behavior. Additionally, one could argue that the model is not conforming to REST's assumptions, [31] treats RESTful Web services procedurally, as "flat" entities without resource hierarchy. In [31] and originally [40] SA-REST is introduced as an extension to provide semantic annotations for API descriptions. From the point of view of modelling RESTful behavior it has little value.

A similar approach to hRESTS is WADL (*Web Application Description Language*) [23]. It is machine-readable (XML) format to describe all kinds of applications

that use HTTP protocol as its transport layer. In this terms it is also suitable to describe APIs of RESTful Web services implemented in HTTP but also ROA systems, which exclusively use HTTP as the communication medium. WADL does not define any model of behavior but in contrary to hRESTS is more RESTful oriented, i.e. it uses the notion of resources and sub-resources, therefore it allows to directly model hierarchy.

The last article that tries to approach a problem to model REST is [46]. The article does not try to provide a more expressive model for RESTful and ROA Web services. Authors use existing CCS calculus to make a model of a RESTful architecture, i.e. The contribution of the article can be treated as a template to create models of RESTful based systems in order to carry out verification and analysis.

## 8 Concluding remarks and guidelines for future REST and ROA process calculi

REST and ROA are promising paradigms, enabling lightweight SOA systems, created according to the existing Web technologies. In this paper we have showed how REST and ROA differ from, SOAP-based Web Services. We have depicted the differences in basic abstractions of the both approaches, such as granularity of the system, importance of semantics of messages, and the supported communication schemas.

In the second part of the article we analyzed process calculi for SOAP-based systems that can be used to model such systems in formal way. Process calculi models are a well known behavioral technique, which enables the modeler to formally analyze, and verify the behavior of a system. This also opens possibilities for automatic and semi-automatic business process composition and finally can be used in a new and booming research ares, process mining (service mining to be exact), and conformance checking [41, 42, 15]. Since there are no process calculi for REST and ROA yet, we decided to focus on behavioral formalisms created specifically for SOAP-based systems: CaSPiS, SOCK, CC, SCC, and COWS. We have proposed a comparison framework, in which we highlighted seven categories in which process calculi have been compared: communication, composition and hierarchy modeling, tool support, equivalences and formal correctness, goals and domain specificness. In each of the categories, every calculus has been discussed and analyzed. We have focused on features that could be helpful when modeling REST and ROA systems but also we have tried to depict features that are not compatible with REST, and would render models in which REST specific characteristics would be lost.

Additionally we have briefly discussed three articles that tackle the problem of formalization of RESTful Web services systems. One of the introduces a very promising formal model that can be used as a base for a future process calculus for REST and ROA. The second article introduces process calculus for semantic REST but it lacks support for most of features that make REST and ROA distinguishing. Finally, the third article uses an existing process calculus just to model so called REST architecture, so its contribution is more a case study than a new formalism.

Based on the analysis in the article, we state that the existing process calculi

for SOAP-based systems are not fully suitable to be used to model REST and ROA systems. There is a need for a formalism specifically designed for the RESTful Web services and ROA architecture. Some ideas and features of the discussed calculi can be however used or provide an inspiration.

The first requirement for a REST process calculus is communication compatible with REST and ROA. Since the number of operations in ROA is finite and limited to basic CRUD operation, mapped onto HTTP methods, the new calculus for REST should define semantics for these messages directly. This means that some messages have different effects on a called resource because of the idempotence, and safety constrains. This would allow an explicit modeling of messaging in REST, where a message type carries the semantics of the request. On top of that messaging schemas, as described in Section 2, could be provided, either directly in semantics of the new language or as syntax patterns. The third requirement for a new REST calculus is the support for expressing hierarchy and orchestration. We feel that a direct approach to defining how elements of the model are composed is superior to the one, where the hierarchy and composition can be only extracted implicitly. Having a separate system equation allows for more flexible modeling and such a calculus can be easier encoded as a practical tool. One of the things that should also be considered in the new formalism is a possibility to express different resource representations and links among resources. We feel that this could be quite easily expressed by allowing for variables and data passing. The final requirement is that the new calculus for REST should be equipped with a suitable bisimulation based equivalence relation. Not every calculus for SOAP-based systems discussed in this article even approached this problem. Equivalences allow to compare processes and determine whether two of them are equivalent in behavior, further this allows to automatically exchange one process with another, and even use methods of reducing model complexity. We feel that classic bisimulation could be suited to REST and ROA systems in order to catch different aspects of processes.

In our future work we would like to address the problem of creating a new process calculus for REST, which will fulfill all the requirements stated above. Out aim is to provide a powerful formal tool, which is from one point is easy to use by engineers, and from another provides a rigor of formal methods.

## Acknowledgements

This work was supported by the Polish National Science Center under Grant No. DEC-2012/05/N/ST6/03051.

## References

[1] van der Aalst, W.: Service mining: Using process mining to discover, check, and improve service behavior. IEEE Transactions on Services Computing

99(PrePrints), 1 (2012)

[2] van der Aalst, W.M., Ter Hofstede, A.H., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and parallel databases 14(1), 5–51 (2003)

[3] Baretto, C., Bullard, V.: Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.html (2007)

[4] Bergstra, J.A.: Handbook of Process Algebra. Elsevier Science Inc., New York, NY, USA (2001)

[5] Bettini, L., De Nicola, R., Loreti, M.: Implementing session centered calculi. In: Proceedings of the 10th International Conference on Coordination Models and Languages. pp. 17–32. COORDINATION'08, Springer-Verlag, Berlin, Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1788954.1788956

[6] Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., et al.: SCC: a service centered calculus. In: Web services and formal methods, pp. 38–57. Springer (2006)

[7] Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and pipelines for structured service programming. In: Formal Methods for Open Object-Based Distributed Systems, pp. 19–38. Springer (2008)

[8] Bravetti, M., Gilmore, S., Guidi, C., Tribastone, M.: Replicating web services for scalability. In: Trustworthy Global Computing, pp. 204–221. Springer (2008)

[9] Brzeziński, J., Danilecki, A., Flotyński, J., Kobusińska, A., Stroiński, A.: ROsWeL Workflow Language: A Declarative, Resource-oriented Approach. New Generation Computing 30(2–3), 141–164 (2012)

[10] Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Towards a formal framework for choreography. In: In Proc. of 3rd International Workshop on Distributed and Mobile Collaboration (DMC 2005). IEEE Computer. pp. 107–112. Society Press (2005)

[11] Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration conformance for system design. In: Coordination Models and Languages. pp. 63–81. Springer (2006)

[12] Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration conformance for system design. In: Coordination Models and Languages. pp. 63–81. Springer (2006)

[13] Caires, L., Vieira, H.T.: SLMC: a tool for model checking concurrent systems against dynamical spatial logic specifications. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 485–491. Springer (2012)

[14] Dustdar, S., Gombotz, R., Baina, K.: Web Services Interaction Mining. Tech. Rep. TUV-1841-2004-16, Technical University of Vienna, Information Systems Institute, Distributed Systems Group (2004)

[15] Dwornikowski, D., Stroiński, A., Brzeziński, J.: Conformance Checking of Communicating Resource Systems with RAs Calculus. In: Services Computing (SCC), 2015 IEEE International Conference on. pp. 759–764. IEEE (2015)

[16] Emerson, E.A., Clarke, E.M.: Characterizing correctness properties of parallel programs using fixpoints. Springer (1980)

[17] Fielding, R., J., R.: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content (2014)

[18] Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)

[19] Fokkink, W.: Introduction to Process Algebra. Springer (2000)

[20] Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems (TOPLAS) 7(1), 80–112 (1985)

[21] Guidi, C., Lucchi, R.: Mobility mechanisms in service oriented computing. In: In: Proc. of 8th International Conference on on Formal Methods for Open ObjectBased Distributed Systems. pp. 233–250. Springer (2006)

[22] Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: SOCK: a calculus for service oriented computing. In: Service-Oriented Computing–ICSOC 2006, pp. 327–338. Springer (2006)

[23] Hadley, M.J.: Web application description language (wadl). Tech. rep., Sun Microsystems,Inc., Mountain View, CA, USA (2006)

[24] He, H.: Implementing REST Web Services: Best Practices and Guidelines. http://www.xml.com/pub/a/2004/08/11/rest.html (2004)

[25] Hernández, A.G., García, M.N.M.: A formal definition of RESTful semantic web services. In: Proceedings of the First International Workshop on RESTful Design. pp. 39–45. ACM (2010)

[26] jolie-lang.org: JOLIE: homepage. http://www.jolie-lang.org/ (2013)

[27] JOpera.org: JOpera - process support for Web Services. http://www.jopera.org/ (2013)

[28] Juric B., M.: A Hands-on Introduction to BPEL (2006), oracle 2006

[29] Kitchin, D., Quark, A., Cook, W., Misra, J.: The Orc programming language. In: Formal Techniques for Distributed Systems, pp. 1–25. Springer (2009)

[30] Klein, U., Namjoshi, K.S.: Formalization and Automated Verification of RESTful Behavior. In: Computer Aided Verification. pp. 541–556. Springer (2011)

[31] Kopecký, J., Gomadam, K., Vitvar, T.: hrests: An html microformat for describing restful web services. In: Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Volume 01. pp. 619–625. WI-IAT '08, IEEE Computer Society, Washington, DC, USA (2008), http://dx.doi.org/10.1109/WIIAT.2008.379

[32] Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web servuces (full version). Tech. rep., Univ. Firenze (2006)

[33] Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: Programming Languages and Systems, pp. 33–47. Springer (2007)

[34] Lathem, J., Gomadam, K., Sheth, A.P.: Sa-rest and (s)mashups: Adding semantics to restful services. In: Proceedings of the International Conference on Semantic Computing. pp. 469–476. ICSC '07, IEEE Computer Society, Washington, DC, USA (2007)

[35] Pautasso, C., Zimmermann, O., Leymann, F.: Restful web services vs. big'web services: making the right architectural decision. In: Proceedings of the 17th international conference on World Wide Web. pp. 805–814. ACM (2008)

[36] Plotkin, G.: A structural approach to operational semantics (1981)

[37] Pnueli, A.: The temporal logic of programs. In: Foundations of Computer Science, 1977., 18th Annual Symposium on. pp. 46–57. IEEE (1977)

[38] Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly Media (2007)

[39] Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press, New York, NY, USA (2011)

[40] Sheth, A.P., Gomadam, K., Lathem, J.: Sa-rest: Semantically interoperable and easier-to-use services and mashups. IEEE Internet Computing 11(6), 91–94 (Nov 2007), http://dx.doi.org/10.1109/MIC.2007.133

[41] Stroiński, A., Dwornikowski, D., Brzeziński, J.: Resource Mining: Applying Process Mining to Resource-Oriented Systems. In: Business Information Systems, pp. 217–228. Springer International Publishing (2014)

[42] Stroiński, A., Dwornikowski, D., Brzeziński, J.: RESTful Web Service Mining: simple algorithm supporting resource-oriented systems. In: Web Services (ICWS), 2014 IEEE International Conference on. pp. 694–695. IEEE (2014)

[43] Van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. Journal of the ACM (JACM) 43(3), 555–600 (1996)

[44] Vieira, H.T., Caires, L., Seco, J.C.: The Conversation Ccalculus: A model of Service Oriented Computation. Tech. Rep. TR-DI/FCT/UNL 6/07, Universidade Nova de Lisboa (2007)

[45] Vieira, H.T., Caires, L., Seco, J.C.: The conversation calculus: A model of service-oriented computation. In: Programming Languages and Systems, pp. 269–283. Springer (2008)

[46] Wu, X., Zhang, Y., Zhu, H., Zhao, Y., Sun, Z., Liu, P.: Formal Modeling and Analysis of the REST Architecture Using CSP. In: Web Services and Formal Methods, pp. 87–102. Springer (2013)

*Received 11.05.2015, accepted 06.10.2015*