A Language for Microservices Interactions*

Chiara Bodei[1], Linda Brodo[2], and Roberto Bruni[1]

[1] University of Pisa, Italy
[2] University of Sassari, Italy

Abstract

Microservices are independent, fine-grained, cohesive software components that communicate via message passing. They have been particularly successful to refactor large service applications in order to improve their scalability. Here we propose a process algebraic framework for formally reasoning on the composition of microservices. In particular, given a microservice system, we can analyse its communication schema and synthesise the corresponding orchestrators.

1 Introduction

Microservices are the current incarnation of the Service Oriented Computing (SOC) paradigm [3]. The SOC principle is to build applications by composing loosely coupled units (services), made available over the network. The microservices architecture [10] pushes the idea further by reducing the size of services into the smallest independent units that implement some related functionalities (cohesion). Microservices do not share memory and can only interact via message passing (service invocation), also synchronously (request-response).

The realisation of distributed application involves (micro)service interaction to exchange data. There are two mainstream approaches to (micro)service composition, called orchestration and choreography [15]. Orchestration is about describing and executing a single view point model: a central server takes care of invoking the services and collecting their outcomes. The analogy is with the conductor who centrally controls the musicians in an orchestra.

Choreography is instead about describing a global model of the interactions between peers: as such a choreography is not executable, but, via endpoint projections, it is possible to derive correct-by-design executable implementations (multi-party session types [11] and choreographic programming [13]). The analogy is with the dancers who behave autonomously, but follow their parts in the choreography

The above features of microservices require programming languages whose design has been targeted to microservices aggregation. One prominent example is Jolie [14, 1]. It offers an imperative core enriched with constructs for distribution and communication, whose semantics is formalised as a process calculus.

Here, we take a slightly different perspective to microservice composition, by allowing *multiparty interaction abstractions* as atomic activities, to be executed in the whole, similarly to [9]. We believe that in many situations this is a much more convenient abstraction for the programmer than the classical binary interaction, because it allows focussing on the overall communication logic of those atomic steps and not on the coordination of all low-level interactions. The overall idea is to have a formal model that allows designers of microservices architectures to logically reason on transactions and their communication properties. In particular, it is important to understand the security challenges posed by the microservice paradigm.

*Research supported by MIUR PRIN 201784YSZ5 *ASPRA: Analysis of program analyses*, by PRIN 2017FTXR7S *IT MATTERS: Methods and Tools for Trustworthy Smart Systems*, by Univ. of Pisa PRA_2018_66 *DECLWARE*, and by Univ. of Sassari *Fondi di Ateneo per la ricerca 2019.*

Our proposal, called `microlink`, builds on the `link`-calculus [5, 6], a multiparty extension of the π-calculus [12]. In the spirit of process calculi, the idea is to define a mathematical framework for high-level descriptions of the behaviour of microservice systems. These descriptions can thus be formally analysed and manipulated in order to reason on the systems properties.

2 `link`-calculus

The `link`-calculus [4] uses the classical operators of process calculi such as action prefix $\alpha.P$, choice $(P + Q)$, parallel composition $(P|Q)$, etc. Its distinguishing feature is a primitive interaction mechanism that offers two general abstractions: multiparty communication and bidirectional information exchange. In its more general version [7], actions $\alpha = vt$ involve chains v of links that describe the communication routing hop by hop. Furthermore, actions also carry a polyadic communication tuple t, whose elements can be used as input or as output names. Each party offers its chain and its tuple, and a multiparty interaction is possible when chains and tuples can fit together.

As in π-calculus, the operational semantics can be defined in the SOS style [16], to derive a Labelled Transition System. Since the `link`-calculus offers multiparty interactions, the main difference with respect to π-calculus is that a communication is completed when all the participants provide their mergeable link chains and tuples.

The `link`-calculus web page [2] collects all the information on the calculus, together with some formal analysis techniques [8], and a reference to the SilVer tool to model check processes.

Links and chains. A *link* ℓ can be *solid* or *virtual*. A solid link is a pair ${}^{a}\backslash_{b}$, where a and b are channels that record the source and the destination ends of each hop of a communication. A virtual link is written as ${}^{\Box}\backslash_{\Box}$, where $\Box$ represents a missing end. Two links ℓ_1 and ℓ_2 can be juxtaposed, if one of them is virtual or they are both solid and where the destination of ℓ_1 coincides with the source of ℓ_2. A *chain* is a finite juxtaposition $v = \ell_1 ... \ell_n$ of links ℓ_i. Intuitively, links are interlocked in a chain by joining their source and destination ends, by exploiting the virtual links, like pieces in a fragment of a jigsaw puzzle. Two chains v_1 and v_2 can be *merged*, if they are to some extent "compatible", in the sense that: (i) each of them only provides solid links that are missing in the other, and (ii) superimposed together they still form a chain. Positions occupied by virtual links in both chains remain virtual.

For example, $v_1 = {}^{\tau}\backslash^{\Box}_{a}\backslash^{b}_{\Box}\backslash_{\tau}$ can be merged with $v_2 = {}^{a}\backslash_{b}$, to form ${}^{\tau}\backslash^{a}_{a}\backslash^{b}_{b}\backslash_{\tau}$. Instead, the two chains $v_1' = {}^{\tau}\backslash^{\Box}_{a}\backslash^{b}_{\Box}\backslash_{\tau}$ and $v_2' = {}^{a}\backslash_{c}$ cannot be merged, because c does not match with b.

Tuples. Name passing mechanism is based on tuples. Tuples are finite lists $t = \langle w_1, ..., w_n \rangle$, where channel names can be used either as values or as variables (in this case they are underlined). During communication, variables are bound by the provided actual values: names can be combined with input variables of other tuples or possibly matched with their output names.

Two tuples t_1 and t_2 can be *merged*, when the values in matching positions are compatible. Consider, e.g. the two tuples $\langle id, n, \underline{x} \rangle$ and $\langle id, \underline{y}, m \rangle$, with the variables x and y and the values id, n and m. The two tuples can be merged, because they coincide on the first parameter id, the variable x can be bound to the value n, while y can be bound to m: the result is therefore $\langle id, n, m \rangle$.

3 `microlink`: tailoring `link`-calculus for microservices.

A microservice system can be thought as a pool of concurrent interacting processes that cooperate for providing a complex service. As we said, the `link`-calculus communication mechanism requires the peers can form a chain of interactions with mutual data exchange. At the abstract level, the order in which links are added to the chain is not important and each peer can act both as the sender of some data and as the receiver of other data.

Consider the classical and well-known travel booking scenario, where: (i) the customer C asks for a travel package; (ii) the travel aggregator T accepts the request and asks two online search engines, one for the flight F and one for the hotel H, for solutions; (iii) the search engines for flights and hotels provide some alternatives. We can model this scenario in the `link`-calculus, by defining a parallel composition of processes, one for each party:

$$
\begin{array}{lcl}
C & \triangleq & {}^{travel}\backslash_{travel}\langle reqID, \underline{x_{flight}}, \underline{x_{hotel}}\rangle.C' \\
T & \triangleq & {}^{\tau}\backslash^{\Box}_{travel}\backslash^{travel}_{\Box}\backslash^{\Box}_{flight}\backslash^{flight}_{\Box}\backslash^{\Box}_{hotel}\backslash^{hotel}_{\Box}\backslash_{\tau}\langle \underline{x_{req}}, \underline{x_{flight}}, \underline{x_{hotel}}\rangle.T' \\
F & \triangleq & {}^{flight}\backslash_{flight}\langle \underline{x_{req}}, KLM305, \underline{x_{hotel}}\rangle.F' \\
H & \triangleq & {}^{hotel}\backslash_{hotel}\langle \underline{x_{req}}, \underline{x_{flight}}, Thuy\rangle.H'
\end{array}
$$

The dot represents the usual action prefix operator, where C', T', F' and H' represent some suitable continuations. Here, an action prefix has either the form ℓt (a participant) or vt (the orchestrator), for ℓ a single link, v a chain and t a data tuple, possibly with variables $\underline{x}$, waiting for actual values. Each process contributes with its links and data to the overall interaction. When executed, all the links can be joined together in a single transition label as their ends match pairwise, forming the following solid (i.e. without non specified actions $\Box$) chain

$$
{}^{\tau}\backslash^{travel}_{travel}\backslash^{travel}_{travel}\backslash^{flight}_{flight}\backslash^{flight}_{flight}\backslash^{hotel}_{hotel}\backslash^{hotel}_{hotel}\backslash_{\tau},
$$

where (see the coloured pdf version of this paper) the items in black are the answers of the search engines, the one in red is the activity of the travel aggregator and the one in blue is the communication of the customer. At the data level, C offers the request ID to all peers, each search engine offers its value ($KLM305$, $Thuy$) which is taken as input from T and C: when the tuples are combined we get $\langle reqID, KLM305, Thuy\rangle$ with the value $reqID$ ($KLM305$, $Thuy$, resp.) in the place of the variable x_{req} (x_{flight}, x_{hotel}, resp.).

One drawback of this model is that any data is visible to all peers, because the interaction manipulates a unique data tuple. For example, in order to contribute to the interaction, the flight process must format the message by including a field for the hotel information (and vice versa for the hotel). Otherwise the data tuples would be incompatible. In general, this may lead to information leaks.

To address this issue, assuming the process providing the chain acts as a trusted orchestrator, we can tag each argument of its tuple with a set of indices to represent visibility information. Consequently, the model can be refined as follows:

$$
\begin{array}{lcl}
C & \triangleq & {}^{travel}\backslash_{travel}\langle reqID, \underline{x_{flight}}, \underline{x_{hotel}}\rangle.C' \\
T & \triangleq & {}^{\tau}\backslash^{\Box}_{travel}\backslash^{travel}_{\Box}\backslash^{\Box}_{flight}\backslash^{flight}_{\Box}\backslash^{\Box}_{hotel}\backslash^{hotel}_{\Box}\backslash_{\tau}\langle \underline{x_{req}}, \underline{x_{flight}}_{\#1,2}, \underline{x_{hotel}}_{\#1,3}\rangle.T' \\
F & \triangleq & {}^{flight}\backslash_{flight}\langle \underline{x_{req}}, KLM305\rangle.F' \\
H & \triangleq & {}^{hotel}\backslash_{hotel}\langle \underline{x_{req}}, Thuy\rangle.H'
\end{array}
$$

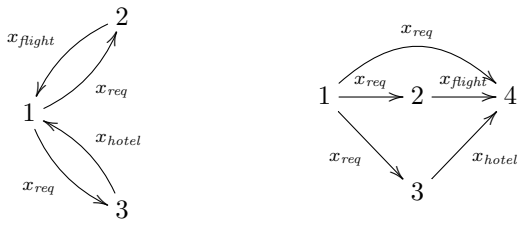

Figure 1: Data dependency graphs

The data tuple represents a temporarily shared memory that collects all the local data updates that are accessible by the orchestrator. For security reasons, each other participant is allowed to view and access only a portion of the tuple, following the access control policy established by the orchestrator. If we imagine that positions in the chain are numbered left to right by consecutive positive naturals starting at 1, syntactically, we can use such numbers as subscripts inside the tuple to mark the arguments available to each peer: in $\langle \underline{x_{req}}, \underline{x_{flight}}_{\#1,2}, \underline{x_{hotel}}_{\#1,3} \rangle$ the information about the flight is shared with the parties in position 1 and 2, the one about the hotel with the ones in position 1 and 3, while the request ID is shared among all the parties (no subscript is necessary in this case). Now, the format of the data tuple provided by the flight does not need to involve the hotel information and vice versa.

In the context of microservices it is important to guarantee the realisability of the orchestration. Data dependencies constrain the order in which services are invoked by the orchestrator. Also the initiator, which invokes the orchestrator, must be determined accordingly. Data visibility information is not enough to establish the choreography, so we need to enrich the above tags by making explicit who is responsible to provide the data and who is only willing to receive or to match it. A possible refinement of the orchestrator is thus the following:

$$T \triangleq \tau\backslash^{\square}_{travel}\backslash^{travel}_{\square}\backslash^{\square}_{flight}\backslash^{flight}_{\square}\backslash^{\square}_{hotel}\backslash^{hotel}_{\square}\backslash_{\tau}\langle \underline{x_{req}}_{\#1\to 2,3}, \underline{x_{flight}}_{\#2\to 1}, \underline{x_{hotel}}_{\#3\to 1}\rangle.T'$$

Here, $\#1 \to 2,3$ means that the information about the $reqID$ is provided from party in position 1 and used by parties in positions 2 and 3. The annotations $\#2 \to 1$ and $\#3 \to 1$ have a similar meaning. The calculus with this new kind of annotations is called `microlink`.

Unfortunately, in our example, if we draw the data dependency graph (see Fig. 1, left) we find out some circularities that prevent one to identify the initiator and establish the (partial) order for service invocation: peer 1 expects some data from 2 and 3 and also provides some data to them.

To remedy this, we can split the client specification in two parts: one acting as the initiator and the other as collector of the response:

$$C \triangleq {}^{travel}\backslash_{travel}\langle reqID\rangle.\mathbf{0} \mid {}^{pack}\backslash_{pack}\langle reqID, \underline{x_{flight}}, \underline{x_{hotel}}\rangle.C'$$

$$T \triangleq \tau\backslash^{\square}_{travel}\backslash^{travel}_{\square}\backslash^{\square}_{flight}\backslash^{flight}_{\square}\backslash^{\square}_{hotel}\backslash^{hotel}_{\square}\backslash^{\square}_{pack}\backslash^{pack}_{\square}\backslash_{\tau}$$
$$\langle \underline{x_{req}}_{\#1\to 2,3,4}, \underline{x_{flight}}_{\#2\to 4}, \underline{x_{hotel}}_{\#3\to 4}\rangle.T'$$

The corresponding dependency graph is in Fig. 1, right: Since the dependency graph is acyclic, it can be used to derive, in turn, the orchestrator for the travel aggregator. We introduce the peer 0 to represent the orchestrator. For any node n in the graph, (i) if n has only outgoing arcs, there is a one-way communication from n to 0, written as $n \to 0$; (ii) if n has incoming

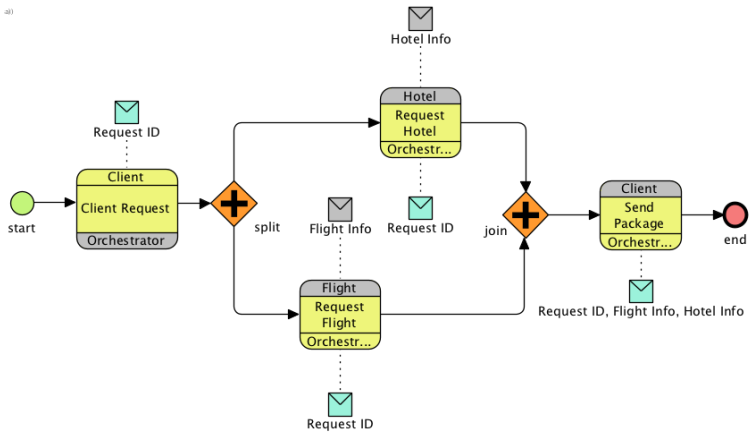

Figure 2: A BPMN choreography diagram for the travel aggregator system.

arcs, there is a communication from 0 to n; (iii) if n has also some outgoing arcs, then the communication between 0 and n is of the kind request-response, written as $n \rightleftarrows 0$. Any order in the communications that is compatible with the dependency graph is acceptable. For example, we can extract an abstract specification of the orchestrator (see below), where:

- the messages associated with *one-way communications* between the orchestrator and a peer n have the form $a\langle t\rangle$, where a is the channel name of peer n and t is the tuple that collects all the labels of the arcs attached to n in the dependency graph;
- the messages associated with *request-response communications* between the orchestrator and a peer n have the form $a\langle t_1\rangle\langle t_2\rangle$, where a is the channel name of peer n, t_1 is the tuple that collects all the labels of the incoming arcs attached to n and t_2 is the tuple that collects all the labels of the outgoing arcs attached to n.

$$
\begin{array}{ll}
1 \to 0: & \mathit{travel}\ \langle x_{req}\rangle;\\
0 \rightleftarrows 2: & \mathit{flight}\ \langle x_{req}\rangle\ \langle x_{\mathit{flight}}\rangle;\\
0 \rightleftarrows 3: & \mathit{hotel}\ \langle x_{req}\rangle\ \langle x_{hotel}\rangle;\\
0 \to 4: & \mathit{pack}\ \langle x_{req}, x_{\mathit{flight}}, x_{hotel}\rangle
\end{array}
$$

In our example, $1 \to 0 : \mathit{travel}\ \langle x_{req}\rangle$ means that the Client invokes the travel aggregator service, by instantiating the variable x_{req}, while $0 \rightleftarrows 2 : \mathit{flight}\ \langle x_{req}\rangle\ \langle x_{\mathit{flight}}\rangle$ means that the orchestrator invokes the request-response operation flight at the Flight service sending the content of the variable x_{req} and storing the response into the variable x_{flight}. Similarly, with $0 \rightleftarrows 3 : \mathit{hotel}\ \langle x_{req}\rangle\ \langle x_{hotel}\rangle$ the orchestrator invokes the request-response operation hotel. Finally, with $0 \to 4 : \mathit{pack}\ \langle x_{req}, x_{\mathit{flight}}, x_{hotel}\rangle$, the orchestrator collects all the information to be sent as an answer to the Client. Note that the order of communications $0 \rightleftarrows 2$ and $0 \rightleftarrows 3$ is irrelevant and they can be executed also in parallel. The above protocol can be conveniently represented as a BPMN choreography diagram[1], as in Figure 2.

Note that this is a very particular kind of choreography, because the orchestrator is involved in every communication and each other peer is instead involved in only one communication.

[1] http://www.bpmn.org

4 Conclusions

We have proposed `microlink`, a language for the high-level specification of microservices interactions, based on the `link`-calculus, where communications require the peers to form a chain of interactions with mutual data exchange. For security reasons, each peer is allowed to view and access only a portion of the data tuple, following the given access control policy. Furthermore, it is important to formally verify the realisability of the communication schema.

Technically, the idea is to annotate each element of the data tuples with the information about which peers can access it and which one provides it. These annotations allows us to extract a dependency graph of data exchange and check whether there are ambiguous cyclic dependencies. When the graph is acyclic, it is possible to further derive a special kind of choreography that includes the orchestrator. Here, the orchestrator takes part in every communication and manages the temporary shared memory of the current transaction. Each other peer only interacts with the orchestrator via a unique one-way or request-response service invocation.

We are currently working on the complete formal specification of `microlink`, to extend tuple elements with expressions. We then would like to develop a tool for the automatic synthesis of the orchestrator.

One research direction for our future work is on some form of failure handling and transaction recovery mechanisms in order to guarantee the correct behaviour of all participants at the end of the sequence of interactions and also to guarantee backup services.

Another direction consists in loosening the central control of the orchestrator, by synthesising a choreography where some peers are allowed to interact without mediation.

References

[1] The Jolie language website. http://www.jolie-lang.org. Visited: August 2020.

[2] The `link`-calculus homepage. http://linkcalculus.di.unipi.it/. Visited: August 2020.

[3] SOA reference model. http://www.oasis-open.org/committees/soa-rm/. Visited: August 2020.

[4] C. Bodei, L. Brodo, and R. Bruni. Open multiparty interaction. In *WADT 2012*, vol. 7841 of *LNCS*, pages 1–23. Springer, 2012.

[5] C. Bodei, L. Brodo, and R. Bruni. A formal approach to open multiparty interactions. *Theor. Comput. Sci.*, 763:38–65, 2019.

[6] C. Bodei, L. Brodo, and R. Bruni. The link-calculus for open multiparty interactions. *Information and Computation*, 2020. To appear.

[7] L. Brodo, R. Bruni, and M. Falaschi. Enhancing reaction systems: A process algebraic approach. In *The Art of Modelling Computational Systems*, vol. 11760 of *LNCS*, pages 68–85. Springer, 2019.

[8] L. Brodo, C. Olarte. Verification techniques for a network algebra. *Fund. Inf.*, 172(1):1–38, 2020.

[9] L. Cruz-Filipe, F. Montesi, and M. Peressotti. Communications in choreographies, revisited. In *SAC 2018*, pages 1248–1255. ACM, 2018.

[10] M. Fowler and J. Lewis. Microservices. ThoughtWorks, 2014.

[11] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL 2008*, pages 273–284. ACM, 2008.

[12] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[13] F. Montesi. *Choreographic Programming*. PhD thesis, 2013.

[14] F. Montesi, C. Guidi, and G. Zavattaro. Service-oriented programming with Jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.

[15] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, Oct. 2003.

[16] G. D. Plotkin. A structural approach to operational semantics. *JLAMP*, 60-61:17–139, 2004.