2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW) | 979-8-3503-3032-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASEW60602.2023.00020

Migrating from monoliths to microservices: enforcing correct coordination

Marco Autili, Gianluca Filippone, Massimo Tivoli
Department of Information Engineering, Computer Science and Mathematics
University of L'Aquila
L'Aquila, Italy
{marco.autili,gianluca.filippone,massimo.tivoli}@univaq.it

***Abstract*—A current trend in service-oriented architectures is to break coarse-grained monolith systems, encapsulating all function capabilities, down into small-scale and fine-grained microservices, which work in concert. The microservices resulting from the decomposition can be independently deployed on physically distributed machines, and an extremely challenging and complex task is to ensure that the behavior emerging from their distributed interaction is equivalent to the original monolith system. Specifically, the price to be paid for the gained distribution is that the emerging microservices interaction may exhibit not only deadlocking behavior, but also extra behavior, which is undesired with respect to the original monolith. In this paper, we propose a method for automatically (i) detecting both deadlocking interactions and extra behavior, and (ii) synthesizing distributed coordinators that when interposed among the resulting microservices avoid deadlocks and undesired interactions.**

***Index Terms*—Microservices Migration, Distributed Coordination, Deadlock Prevention.**

I. Introduction

Service-oriented Architectures (SOA) have established for years as the main approach for building large-scale distributed systems by composing software services in a loosely coupled fashion [1]–[3]. In SOA, services can be reused and composed as black-box entities that interact over the network through a standardized interaction protocol. In recent years, further advancement of SOA consisted of microservice architectures [4]–[6]. By leveraging on the main principles of SOA, microservices are built through the composition of fine-grained, loosely coupled, and autonomous services that are organized around a specific business capability, can be independently deployed, and communicate through lightweight protocols [7], [8].

Problem space – Microservices are beneficial in terms of scalability, flexibility, and resilience, allowing for more efficient use of resources and infrastructure and better support for continuous integration and delivery [7], [9], [10]. In order to take advantage of these benefits, companies are migrating their legacy monolithic services to microservices [11]–[14]. Although, in principle, the microservices resulting from the migration are independent from each other and can be independently deployed on physically distributed machines, they must work together to offer the desired functional behavior. For the purpose of this work, we consider as *desired* a behavior that is equivalent to that of the original monolith system, in terms of allowed sequences of message exchanges. That said, the following statement generalizes the above problem to more than one monolith system, in that it considers a set of interacting monolithic coarse-grained services.

Problem statement – *Given a set of interacting monolithic coarse-grained services and a set of interacting microservices representing their decomposition, automatically derive a set of distributed proxies capable of preventing possible deadlocks and coordinating the distributed interaction so that the behavior emerging from the resulting microservice-based system is equivalent to the original monolith services.* Specifically, a crucial implication of the problem statement is that the resulting microservice-based system may exhibit not only deadlocking behavior but also extra behavior, which is undesired with respect to the original monolithic coarse-grained services.

Before introducing the solution we have in mind, an important note is that, whenever an incremental migration process is being applied (as opposed to migrating everything all at once), the decomposition mentioned in the problem statement above may involve only a few of the monolithic coarse-grained services of the original system (see $S2$ decomposed into the two microservices $\mu S2'$ and $\mu S2''$ in Figure 1), leaving the others undecomposed (see $S1$ in the figure).

Solution space – Following the research line in [14] and [15], in this *short paper*, we propose a methodology to synthesize the set of distributed proxies needed to enforce the desired deadlock-free behavior in the resulting microservice-based system. Specifically, it is a reuse-based approach that conveniently applies the "old wine" of behavior coordination and mediation in [16]–[18] (for which correctness and soundness are formally proven) to the "new bottle" of microservices. More technically, without reinventing the wheel, we reuse the core algorithm for identifying and preventing deadlocks initially proposed in [16] for component-based systems and accommodate it to microservice-based ones.

We make use of automata, in the form of Labeled Transition Systems (LTSs) [19], to describe the message-passing behavior of the involved (micro)services. We exploit the notion of service composition to obtain and model the new microservice-based systems and use it to detect possible deadlocks and check for the enforceability of the desired behavior. Finally, we synthesize the behavior of the distributed proxies to be interposed (when needed) between those coarse-grained services of the original system left undecomposed, and the fine-grained microservices obtained with the decomposition (see P_1, $P'_{\mu 2}$ and $P''_{\mu 2}$ in Figure 1).

The paper is organized as follows. Section II overviews the approach we have in mind, and Section III briefly discusses our future plan to fully formalize it. Section IV describes it at work on an explanatory example. Section V concludes the paper and discusses future work.

II. Approach overview

The approach takes as input the behavioral models of the microservices obtained with the decomposition and those of the original (undecomposed) services (see $\mu S2'$, $\mu S2''$, and $S1$ in Figure 1, respectively). As for the decomposition step, there exist many approaches at the *sota* for decomposing each of the coarse-grained services in a set of microservices offering the monolith's functionalities suitably combined together, both with the assistance of semi-automated [20]–[23] or automated tools [14].

The behavioral models of both coarse-grained and micro-services is obtained by considering them as black-box entities and observing their behavior "externally". The sequence of interactions (i.e., the sending and receiving of messages) that each (micro) service performs with other (micro) services is represented as a LTS that describes the behavior of a service.

From the LTSs of services, the overall system behavior of both (i) the original coarse-grained-based system and (ii) the decomposed microservice-based one are obtained. As clarified in the next section, the system behavior is modeled by composing in parallel the LTSs of every single service in the system (be it an undecomposed coarse-grained service or be it a microservice obtained with the decomposition) and forcing synchronization on common send/receive messages. Running the parallel composition with the original coarse-grained services and with the decomposed microservices allows us to obtain two LTSs describing the behavior of the original and the decomposed system, respectively.

The behavioral models are then analyzed for deadlock detection. Sink states that may emerge in the achieved parallel composition LTSs represent deadlocks states. Deadlock prevention is performed by pruning any possible sink state from the LTSs.

After removing possible deadlocks, the parallel composition LTSs are analyzed in the next step to identify and remove those interaction traces in the "decomposed" system constituting undesired extra behavior in that they are not allowed in the original system. In doing this, LTSs from the original and the decomposed system are compared, and all the interactions from the LTS of the decomposed system constituting traces that are not in the original LTS are removed until the behavior of the decomposed system becomes bisimilar [24] to that of the original one. This process produces a model of the "desired" behavior of the decomposed system, where all and only the sequences of interactions of the original system are allowed.

At runtime, deadlock prevention and desired behavior enforcement call for suitably coordinating those transitions leading to sink states, so as to avoid them before reaching a (deadlocking) "point of no return", as well as those that, if left uncontrolled, would lead the decomposed system to execute undesired execution traces. Specifically, in order to enforce the desired behavior at run time, a set of distributed proxies is automatically synthesized by projecting the desired behavior LTS onto each (micro)service and generating the required distributed coordination logic for each proxy (see P_1, $P'_{\mu 2}$ and $P''_{\mu 2}$ in Figure 1). When interposed among the (micro)services constituting the decomposed system, the synthesized proxies enforce the distributed interaction so that the filtered behavior is deadlock-free and equivalent to the original system. Each proxy receives messages sent by its associated (micro)service and forwards them to the target (micro)service; receives messages from other (micro)services and forwards them to its associated, according to the desired behavior of the system.

III. On the approach formalization

As already anticipated, in order to model the behavior of the interacting (micro)services, we use Labeled Transition Systems (LTSs) [19], [25] (having no accepting state) since, in a black-box setting, what matters about a service interaction is not whether it drives the automaton in an accepting state, rather, whether the service is able to perform the corresponding sequence of actions interactively.

For greater flexibility at the behavior specification level, we consider both deterministic LTSs and non-deterministic LTSs. The notion of *trace* is then used to model the sequences of messages or messages exchanges via synchronization, and a trace-based semantics is associated with the LTSs in order to model the externally observable behavior of (micro)services, i.e., all the possible traces originating from the initial state.

The parallel composition operator "$|$" is considered to combine the behaviors of two or more LTSs. The operator synchronizes shared/common messages and interleaves non-shared messages. Importantly, the proposed approach takes as input only LTSs that do not contain internal actions, and the parallel operator we employ never produces epsilon transitions in the place of synchronized messages; rather, it keeps track of the synchronized messages by retaining transitions whose labels specify the exchanged messages, the sender, and the receiver services. Moreover, a notion of reachability is used so that the isolated parts resulting from the parallel composition (i.e., not reachable from the initial state) are ignored, as they have no semantic significance.

Finally, a notion of strong bisimulation [24], modulo a suitable relabelling of the transitions (that allows to assume a transition $a_{S1 \rightarrow S2}$ equal to $a_{S1 \rightarrow S2'}$, being $S2'$ a service deriving from the decomposition of $S2$), is used to check whether the behavior of the decomposed system is equivalent to that of the original one. Note that, strong bisimulation particularly suites our needs since (*i*) it is an equivalence relation that can be defined also over non-deterministic LTSs,

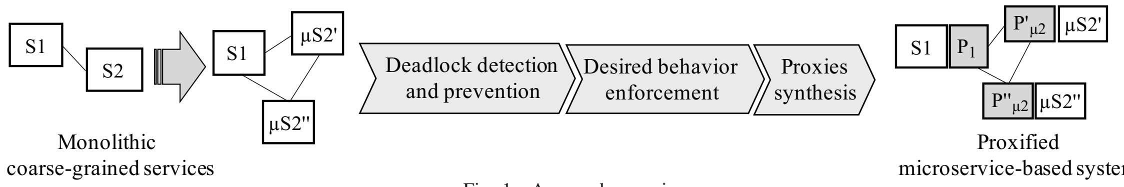

S1
S2
S1
μS2'
μS2''
Monolithic coarse-grained services
Deadlock detection and prevention
Desired behavior enforcement
Proxies synthesis
S1
P1
P'μ2
μS2'
P''μ2
μS2''
Proxified microservice-based system

Fig. 1. Approach overview

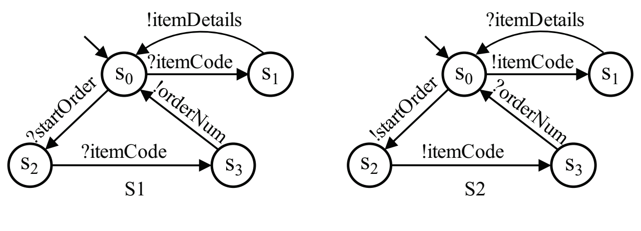


Fig. 2. Behavior of monolithic services `S1` and `S2`

(*ii*) we are modeling interacting systems, (*iii*) we are not interested in unobservable behavior (i.e., internal actions).

IV. Approach at Work

This section describes our approach at work on an explanatory example. The example is very basic, though we have "artificially" modeled a set of interactions that permit us to informally describe the approach at work on interesting cases, which concern both deadlocking behavior and undesired extra behavior. The example considers two very simple services that interact to realize an e-commerce-like system

Figure 2 shows the LTSs of the two monolithic coarse-grained services $S1$ and $S2$ for our example. For the transition labels, we denote with $?a$ (resp., $!a$) the receiving (resp., sending) of the message a. For instance, according to the behavioral specification in the figure, the service $S1$ can either receive the message $itemCode$ and then send the message $itemDetails$, or receive the message $startOrder$ followed by the message $itemCode$ and then send the message $orderNum$.

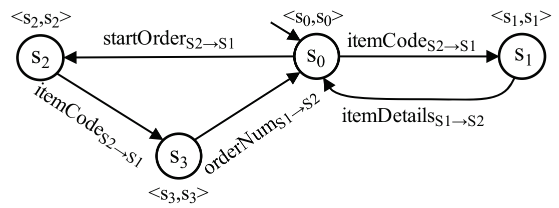


Fig. 3. Parallel composition `S1|S2`

According to the notion of parallel composition in Section III, Figure 3 shows the LTS modeling the interaction behavior of $S1$ and $S2$ ($S1|S2$), via synchronization on common send/receive messages. Thus, $S1$ can synchronize with $S2$ in such a way that $S1$ can either receive the message $itemCode$ from $S2$ ($itemCode_{S2\rightarrow S1}$) and then reply by sending the message $itemDetails$ to $S2$ ($itemDetails_{S1\rightarrow S2}$), or receive the sequence of messages $startOrder$ followed by $itemCode$ from $S2$ ($startOrder_{S2\rightarrow S1}$ followed by $itemCode_{S2\rightarrow S1}$) and then reply by sending the message $orderNum$ to $S2$ ($orderNum_{S1\rightarrow S2}$).

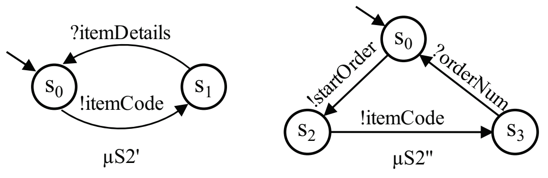


Fig. 4. Behavior of microservices obtained from `S2`

Let us now suppose that the service $S2$ is migrated and decomposed into two microservices $\mu S2'$ and $\mu S2''$ as shown in Figure 4. In this example, microservice $\mu S2'$ now handles the item management of the system, while microservice $\mu S2''$ manages the order creation process. The interaction behavior of the system obtained through the composition $S1|\mu S2'|\mu S2''$ is shown in Figure 5. Note that the composed system now presents an additional state and an additional transition. This means that the behavior of the resulting microservice-based system is different with respect to the original one. In fact, new transitions in the composed LTS mean that there is extra behavior emerging from new interactions, which were not allowed in the original system. Also, the new interactions can lead the system to new states that represent deadlock conditions.

A. Deadlock detection and prevention

In general, a deadlock occurs when two or more transactions are waiting for each other to complete, in such a way that the wait could continue endlessly without any intervention. In our service-oriented setting, we define a deadlock as follows: *a set S of (micro)services is* deadlocked *if each service in S is waiting either for receiving a message that only a different service in S can send or for sending a message that no service in S is able to receive.* According to our interaction model (Section III), when a deadlock occurs, it is observable in the LTS of the parallel composition as a sink state and, for the purpose of this work, we treat it with proxies from the outside (without requiring knowledge of the service's internal working or, equivalently, assuming a black-box setting).

Back to our example, Figure 5 shows the LTS resulting from the parallel composition of the service $S1$ and microservices $\mu S2'$ and $\mu S2''$, representing the behavior of the system having service $S2$ decomposed as described. Here, a deadlock

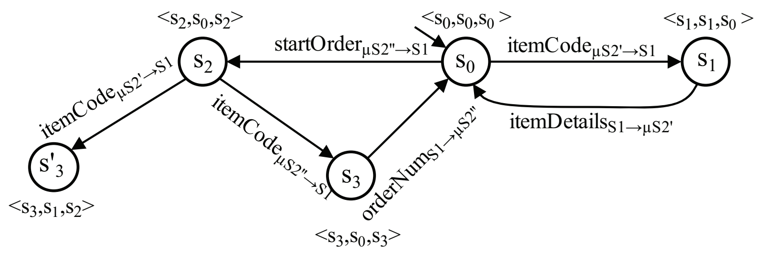


Fig. 5. Parallel composition $S1|\mu S2'|\mu S2''$

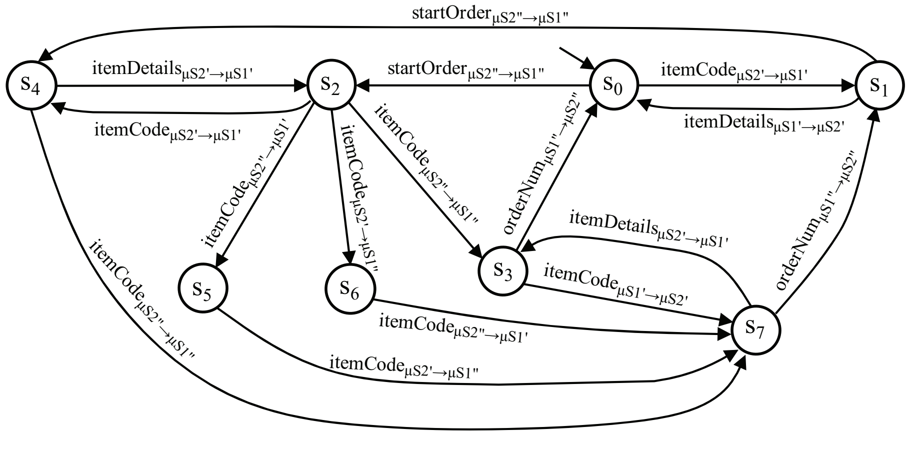


Fig. 6. Parallel composition $\mu S1'|\mu S1''|\mu S2'|\mu S2''$

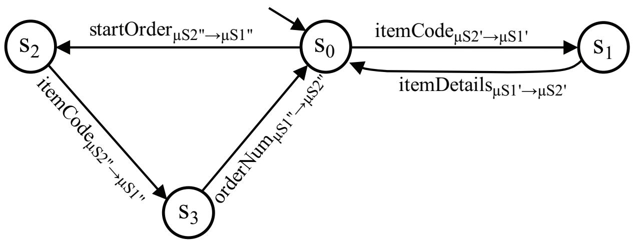
occurs whenever $\mu S2'$ sends the message $itemCode$ to $S1$ just after $S1$ received the message $startOrder$ from $\mu S2''$ (see the deadlocking trace $startOrder_{\mu S2'' \to S1}$ followed by $itemCode_{\mu S2' \to S1}$ in the figure). Specifically, in this case, $S1$ reaches its local state s_3 in which it would send the message $orderNum$ (see Figure 2), but there is no other service ready to receive it, hence blocking its execution. On the other hand, $\mu S2'$ is waiting for receiving the message $itemDetails$ while in its local state s_1 and $\mu S2''$ is ready to send the message $itemCode$ from its local state s_2 (see Figure 4), but there is no other service either sending $itemDetails$ or receiving $itemCode$, hence blocking the execution of $\mu S2'$ and $\mu S2''$. Thus, the sink global state $s_3' = <s_3, s_1, s_2>$ is a deadlock state.

B. Desired behavior enforcement

Now, similarly to what is already done for the service $S2$, let us suppose that also the service $S1$ is decomposed into two microservices $\mu S1'$ and $\mu S1''$. Thus, all the monolithic coarse-grained services of the original system have been decomposed, and their parallel composition $\mu S1'|\ \mu S1''|\ \mu S2'|\ \mu S2''$ is shown in Figure 6 (state quadruples are not shown in the figure for readability). Although there are no sink states, hence no deadlock as in the previous case, the LTS has now a number of additional transitions that are not allowed in the original system, hence showing undesired extra behavior. This means that, for instance, the resulting microservice-based system allows the sequences of messages $(startOrder_{\mu S2'' \to \mu S1''}, itemCode_{\mu S2'' \to \mu S1''}, itemCode_{\mu S2' \to \mu S1'})$, $(startOrder_{\mu S2'' \to \mu S1''}, itemCode_{\mu S2' \to \mu S1''}, itemCode_{\mu S2'' \to \mu S1'})$, and $(itemCode_{\mu S2' \to \mu S1'}, startOrder_{\mu S2'' \to \mu S1''})$ that are not allowed by the original system (see Figure 3) and, as such, represent undesired behavior.

At this point, following the approach overview in Section II, the LTS of the parallel composition of the resulting fully-decomposed system in Figure 6 is "pruned" to remove those transitions leading to sink states and those transitions identifying undesired extra behavior in such a way that it becomes bisimilar [24], modulo a suitable relabelling of the transitions (see Section III), with the LTS of the parallel composition of the monolithic coarse-grained services in Figure 3. The resulting LTS models the desired (global) behavior to be enforced on the decomposed microservice-based system in order to prevent deadlocks and undesired extra behavior.

Figure 7 shows the LTS of the desired behavior for our example. As expected, the only allowed sequences of messages are those that are allowed by the monolithic services shown in Figure 3, modulo suitable relabelling and modulo decomposed services.



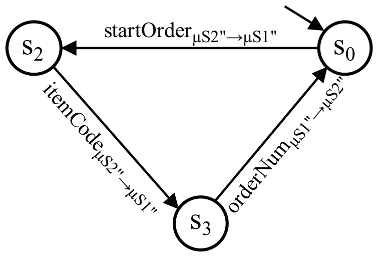
startOrder$_{\mu S2'' \to \mu S1''}$
itemCode$_{\mu S2' \to \mu S1'}$
itemDetails$_{\mu S1' \to \mu S2'}$
itemCode$_{\mu S2'' \to \mu S1''}$
orderNum$_{\mu S1'' \to \mu S2''}$
s_0
s_1
s_2
s_3

Fig. 7. Desired behavior of the resulting microservice-based system

C. Proxies synthesis

As explained, the desired behavior is enforced at runtime through proxies that are interposed between the (micro)services. Their behavior is obtained by projecting the LTS of the system's desired behavior (Figure 7 in our example) onto each participant (i.e., services and microservices) composing the system.

The result of the projection of the desired behavior's LTS on a participant is an LTS that considers only the transitions (and their endpoint states) in which that participant is involved. The LTSs obtained through projection on each participant represent the local models of the desired behavior that has to be enforced by the cooperating local proxies in a fully distributed way.



startOrder$_{\mu S2'' \to \mu S1''}$
itemCode$_{\mu S2'' \to \mu S1''}$
orderNum$_{\mu S1'' \to \mu S2''}$
s_0
s_2
s_3

Fig. 8. Projecion of the behavior on $S2''$

Figure 8 shows the projection of the system's desired behavior (Figure 7) on the participant $S2''$. As explained, it is obtained as a "sub-LTS" that considers only the transactions involving the microservice $S2''$. This model is used to derive the behavior corresponding proxy, which is then synthesized and interposed between the (micro)services to enforce proper coordination.

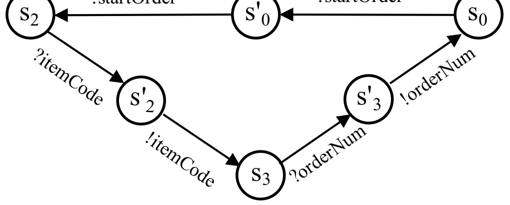


Fig. 9. Behavior of the proxy $P''_{\mu 2}$

Figure 9 shows the behavior of the proxy $P''_{\mu 2}$ obtained from the projection on $S2''$. Each transition in the projected LTS is transformed into two sequential transitions separated by a new "dummy" state. The new transitions are required to let the proxy receive the message and then forward it to the target (micro)service, e.g., the transition $startOrder_{\mu S2'' \to \mu S1''}$ is transformed into the transitions $?startOrder$ and $!startOrder$ with the new state s'_0

Note that, synthesized proxies always have information on which is the target service of each message to be sent, since their behavior is obtained from the projected LTS actually owning this information. For example, from the transition $itemCode_{\mu S2'' \to \mu S1''}$ in the projected LTS (Figure 8), it is derived that the transition $!itemCode$ in the proxy behavior LTS (Figure 9) represents the sending of the message $itemCode$ to $S1''$. This is crucial information that allows proxies to avoid undesired interaction (e.g., sending $itemCode$ message to $\mu S1'$ would result in the extra behavior $itemCode_{\mu S2'' \to \mu S1'}$ from the state s_2 in Figure 6) or incurring in deadlock states.

Proxies are synthesized according to their behavioral model described above (plus the information about the target services of the messages to be forwarded). A model-to-code transformation produces the code of the proxies, that are interposed between participants and enforce the correct (and deadlock-free) behavior of the system.

V. Conclusions

In this short paper, we propose a method to automatically detect both deadlocking interactions and extra behavior, as well as synthesize distributed proxies for coordinating the interaction of a set of microservices obtained by decomposing monolithic coarse-grained services.

The state explosion problem suffered by the parallel composition operator used in our approach is not dealt with in this work. A crucial future step will consider on-the-fly synthesis methods so to mitigate the problem by dynamically building the parallel composition of the overall system without retaining the whole model.

Moreover, we do not consider non-terminating interactions, e.g., due to undelivered messages. As future work, we may introduce timeouts and retry policies to handle and repair undelivered messages.

Another necessary future step will fully formalize the whole approach, extending the formal proofs of correctness and soundness in [16]–[18] to the microservice domain, implement it, and perform validation against real-scale case studies.

Acknowledgment

This work was partially supported by: the Italian Government under CIPE resolution n. 135 (December 21, 2012), project INnovating CIty Planning through Information and Communication Technologies (INCIPICT); the Italian MIUR SISMA national research project (PRIN 2017, Contract 201752ENYB); and Italian PNRR MUR Centro Nazionale HPC, Big Data e Quantum Computing, Spoke9 - Digital Society & Smart Cities.

References

[1] J. Erickson and K. Siau, "Web services, service-oriented computing, and service-oriented architecture: Separating hype from reality," *J. Database Manag.*, vol. 19, no. 3, pp. 42–54, 2008.

[2] E. D. Nitto, C. Ghezzi, A. Metzger, M. P. Papazoglou, and K. Pohl, "A journey to highly dynamic, self-adaptive service-based applications," *Autom. Softw. Eng.*, vol. 15, no. 3-4, pp. 313–341, 2008.

[3] A. Bouguettaya, M. P. Singh, M. N. Huhns, Q. Z. Sheng, H. Dong, Q. Yu, A. G. Neiat, S. Mistry, B. Benatallah, B. Medjahed, M. Ouzzani, F. Casati, X. Liu, H. Wang, D. Georgakopoulos, L. Chen, S. Nepal, Z. Malik, A. Erradi, Y. Wang, M. B. Blake, S. Dustdar, F. Leymann, and M. P. Papazoglou, "A service computing manifesto: the next 10 years," *Communication ACM*, vol. 60, no. 4, pp. 64–72, 2017.

[4] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*, pp. 195–216. Springer, 2017.

[5] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?," *IEEE Software*, vol. 38, no. 1, pp. 32–39, 2021.

[6] L. Baresi and M. Garriga, "Microservices: The evolution and extinction of web services?," in *Microservices, Science and Engineering*, pp. 3–28, Springer, 2020.

[7] J. Lewis and M. Fowler, "Microservices a definition of this new architectural term." https://martinfowler.com/articles/microservices.html, 2014. Accessed on: March 03, 2023.

[8] S. Newman, *Building Microservices*. O'Reilly Media, Inc., 1st ed., 2015.

[9] M. Autili, A. Perucci, and L. D. Lauretis, "A hybrid approach to microservices load balancing," in *Microservices, Science and Engineering*, pp. 249–269, Springer, 2020.

[10] M. Richards, *Software architecture patterns*, vol. 4. O'Reilly Media, Inc., 2015.

[11] P. D. Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *14th IEEE International Conference on Software Architecture (ICSA)*, pp. 21–30, 2017.

[12] D. Wolfart, W. K. G. Assunção, I. F. da Silva, D. C. P. Domingos, E. Schmeing, G. L. D. Villaca, and D. d. N. Paza, "Modernizing legacy systems with microservices: A roadmap," in *Evaluation and Assessment in Software Engineering*, p. 149–159, ACM, 2021.

[13] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann, "Microservices migration in industry: Intentions, strategies, and challenges," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 481–490, 2019.

[14] G. Filippone, N. Q. Mehmood, M. Autili, F. Rossi, and M. Tivol, "From monolithic to microservice architecture: an automated approach based on graph clustering and combinatorial optimization," in *20th IEEE International Conference on Software Architecture (ICSA)*, pp. 1–11, 2023.

[15] G. Filippone, M. Autili, F. Rossi, and M. Tivoli, "Migration of monoliths through the synthesis of microservices using combinatorial optimization," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 144–147, 2021.

[16] M. Autili, L. Mostarda, A. Navarra, and M. Tivoli, "Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems," *Journal of Systems and Software*, vol. 81, no. 12, pp. 2210–2236, 2008.

[17] M. Autili, P. Inverardi, and M. Tivoli, "Choreography realizability enforcement through the automatic synthesis of distributed coordination delegates," *Science of Computer Programming*, vol. 160, pp. 3–29, 2018.

[18] M. Autili, P. Inverardi, R. Spalazzese, M. Tivoli, and F. Mignosi, "Automated synthesis of application-layer connectors from automata-based specifications," *Journal of Computer and System Sciences*, vol. 104, pp. 17–40, 2019.

[19] D. Taubner, "Finite representations of ccs and tcsp programs by automata and petri nets," in *Lecture Notes in Computer Science*, 1989.

[20] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *Service-Oriented and Cloud Computing* (M. Aiello, E. B. Johnsen, S. Dustdar, and I. Georgievski, eds.), (Cham), pp. 185–200, Springer International Publishing, 2016.

[21] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, and Z. Shan, "A dataflow-driven approach to identifying microservices from monolithic applications," *Journal of Systems and Software*, vol. 157, p. 110380, 2019.

[22] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service candidate identification from monolithic systems based on execution traces," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 987–1007, 2021.

[23] L. Baresi, M. Garriga, and A. D. Renzis, "Microservices identification through interface analysis," in *European Conference on Service-Oriented and Cloud Computing*, pp. 19–33, Springer, 2017.

[24] R. Milner, *Communication and concurrency*, vol. 84. Prentice hall Englewood Cliffs, 1989.

[25] L. Aceto, W. Fokkink, and C. Verhoef, "Structural operational semantics," in *Handbook of Process Algebra* (J. Bergstra, A. Ponse, and S. Smolka, eds.), pp. 197–292, Amsterdam: Elsevier Science, 2001.