Design-time to Run-time Verification of Microservices Based Applications
(Short Paper)

Matteo Camilli, Carlo Bellettini, Lorenzo Capra

Dept. of Computer Science,
Università degli Studi di Milano, Milan, Italy
{camilli,bellettini,capra}@di.unimi.it

Abstract. Microservice based architectures have started to gain in popularity and are often adopted in the implementation of modern cloud, IoT, and large-scale distributed applications. Software life cycles, in this context, are characterized by short iterations, where several updates and new functionalities are continuously integrated many times a day. This paradigm shift calls for new formal approaches to systematic verification and testing of applications in production infrastructures. We introduce an approach to continuous, *design- to run-time verification*, of microservice based applications. This paper describes our envisioned approach, the current stage of this ongoing work, and the challenges ahead.

Keywords: microservices, cloud applications, formal verification, formal methods @ runtime, Petri nets.

1 Introduction and Background

The microservice architectural style [1] represents an upward trending approach to the development of modern cloud, IoT, or more in general large-scale distributed application. Services implement individual functional areas of the application and they may be written using different programming languages and technologies. Moreover, they are independently deployable by automated procedures. This approach has been proposed to cope with many problems associated with monolithic applications, especially as more applications are being deployed into cloud platforms [2]. Among all the issues associated with monolithic products, some notable examples are: hard maintainability; scalability issues; technology lock-in for developers; and expensive delivery of latest builds.

The microservice style is not novel or innovative: it is inspired by service-oriented computing [3] and we can find its roots in the design principles of Unix [2]. However, the shift towards microservices is a sensitive matter nowadays. In fact, several companies are switching to this paradigm by applying major refactoring activities. Netflix, Inc. [4] is a leading example: they recently moved from their previous monolithic application to a microservices architecture with hundreds of services working together to stream multimedia contents to millions of users every day. The whole architecture builds on Netflix Conductor

engine [5], an open source framework designed by Netflix Inc. and used daily in their production environment. Conductor allows the creation of complex process flows in which individual tasks are implemented by microservices. The process flow blueprint is defined using a JSON based DSL and includes a set of *system* tasks (e.g., fork, join, conditional, etc.) executed by Conductor's engine, and *worker* tasks (e.g., file encryption) that are the functional areas of the application, running on remote machines. In this context, formal verification and testing activities can be challenging. In fact, rapidly evolving services potentially require formal models and tests to be recreated or modified. Moreover, microservices' polyglot nature potentially requires multiple testing tools because of different programming languages and runtime environments.

To deal with these open problems, we introduce an approach to formal design-to run-time verification (RV) of microservice-based process flows built on top of Conductor. In particular, we have automated the modeling phase by mechanically translating the Conductor blueprint into a formal specification given in terms of Time Basic Petri nets [6] (from now on simply TB nets). TB nets are an expressive time-extension of Petri nets (PNs) provided with a clear and rigorous semantics, and represent an effective formal specification of distributed systems with time constraints. The TB nets formalism is nicely supported by powerful off-the-shelf software tools covering both modeling and verification phases [7, 8]. The RV technique is currently implemented as a prototypal Java library, built on top of MahaRAJA [9], a lightweight pluggable tool supporting the verification of behavioral and temporal aspects of Java programs.

The paper is organized as follows. In Section 2, we introduce our proposed approach, pointing out the current stage of this ongoing work. We report some related work in Section 3, finally we draw our conclusions, discussing the challenges ahead, in Section 4.

2 Overview of the Approach

Figure 1 shows the two main phases of our approach: (*i*) model generation and verification; and (*ii*) runtime verification. A description of the two phases follows. As a running example, used throughout the discussion, we use a taxi-hailing application, such as Uber [10]. Each microservice implements a particular functional area of the application (e.g., access control, passenger management, trip management, payment, etc.) and exposes a REST API to other microservices or the clients of the application. For example, the *passenger management* service uses the *notification* service to notify a passenger about an available driver. An API gateway exposes a public API used by mobile clients or web UIs. We assume that the process flow of the application is specified with Conductor and is deployed on a cloud platform running the Conductor's engine.

(*i*) Model generation and offline verification – This phase tries to deal with the rapid evolution of microservice systems by formalizing the Conductor blueprint as a TB nets model, describing both the system under development

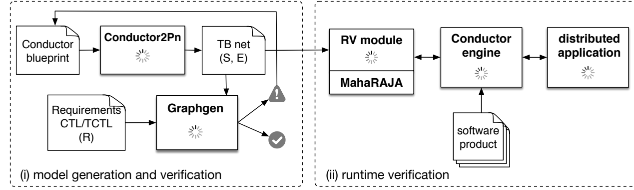

Fig. 1: High level schema of our envisioned approach.

(S) and the environment (E). Every time a change is made to the process flow, the formal specification is automatically kept in sync. The translation process is fully automated by means of the CONDUCTOR2PN component. Our modeling approach abstracts from functional aspects and looks at each service as a black box. We use TB net places to represent the state of a worker task (i.e., scheduled, in progress, timed out, or failed) and transitions to represent both task primitives and events coming from the surrounding environment. We leverage TB nets' temporal functions associated with transitions to specify temporal constraints on scheduling and execution. The proposed formal semantics is complete, i.e., covers all the language constructs of CONDUCTOR.

As an example, the CONDUCTOR blueprint of the taxi-hailing application contains different worker tasks, among which: *Access control* (validating user requests); *Passenger management* (storing and processing passengers data); *Driver management* (looking for available drivers near passengers). System tasks[1] define the process flow. For instance, an *event handler* can be used to elaborate incoming user requests; a *fork* task can be used to execute in parallel the *Access control* functionality and a search for static contents in a *cache* service.

The model generation process is fully compositional, relying on the identification of translation patterns for the involved microservices (Worker tasks) and the execution flow (specified by System tasks). The final model is the composition of the TB net patterns of the microservices and execution flow constructs. It can be used to perform different verification steps, such as interactive simulation and model checking, with the aid of GRAPHGEN module. A common property to check is deadlock/livelock absence. More generally, it is possible to verify *invariant*, *safety*, *liveness* and *bounded-response time* properties corresponding to TCTL formulas [11]. For instance, we can verify whether it is possible to complete the payment and the billing processes within a certain time bound.

***(ii)* Runtime verification** – A model (re)generated in the previous step can be also used to perform verification activities at runtime, on the production infrastructure (the CONDUCTOR engine running on a cloud platform). The objective is to run and monitor several executions in order to stress the system under scrutiny (SUS) and increase our confidence about its correctness.

[1] A list of all the available system tasks can be found in [5].

The RV module exploits the MahaRAJA open source tool [9]. This framework supplies the ability to map methods of interests (i.e., *action methods*) to TB net transitions. At runtime, the MahaRAJA framework performs a monitoring activity through the co-execution of the formal specification (triggered by running action methods) and the SUS. The monitor continuously evaluates the conformance of the execution timed trace, observed from the SUS, with respect to its formal specification. Since the Conductor engine is written in Java, the MahaRAJA framework can be directly plugged-in to monitor at runtime its own execution. As an example, different executions of the taxi-hailing application can be stimulated by issuing user requests, such as: trip computation for a given ⟨source, destination⟩ pair, by a passenger user; pick-up acceptance, by a driver user. Thus, we verify that the involved services act as expected, within the temporal constraints defined in the formal specification, by comparing the observed execution traces with feasible execution paths of the TB net model.

In addition, the RV module can be tuned to distinguish operations under the control of a tester (i.e., *controllable* action methods) from operations that can only be observed (i.e., *observable* action methods). If the running system is in a state of *quiescence* [12], a legal *controllable* action (e.g., a user request in our taxi-hailing example) is randomly chosen by applying a user-defined strategy (based on either fixed or decrementing weights [12]), to automate user interaction. In order to help assess the quality of the runtime verification activity, we allow the user to check if certain requirements (i.e., *goal states*, expressed as reachable TB net markings) are met and we gather some coverage information in terms of executed controllable/observable action methods.

Current Stage of the Work – The model generation is implemented by the Conductor2Pn [13] Java tool. We validated our approach by translating a variety of benchmarking examples. We are currently in the process of trying it out on real-world applications. The generated model can be formally verified using the Graphgen model checker [7, 8, 14]. The RV module, built on top of MahaRAJA [9], is a prototypal implementation and has not yet been fully validated. So far, it has been employed to monitor the execution of the taxi-hailing application running on a locally simulated environment. This has permitted us to discover a number of errors both in the blueprint and the implementation. Most conceptual errors were early discovered at the design-time, what dramatically reduced the number of bugs discovered during testing and RV.

3 Related Work

Although descriptive formalisms for distributed systems are very popular, the adoption of operational specifications offers some advantages with respect to declarative ones [15]. In fact, most operational models are visual and are usually easier to write and understand by non expert users. Automata-based formalisms support the specification of both behavioral and temporal aspects, but PN-based models are more concise and scalable [16]. Moreover, aspects like messaging and communication protocols, commonly used in service oriented and

microservice-based architectures, are difficult to model using the language primitives of automata-based formalisms [17].

The automatic model generation, during the design phase, is somehow inspired by previous studies on Business Process Execution Language (BPEL) for Web Services [18]. However, these approaches cannot be directly applied in the context of microservices, where emerging new languages and frameworks, such as Conductor, are being adopted as major references for orchestration.

Jolie [19] is an interpreter engine of microservice workflows specified by using a Java-like syntax orchestration language. The supplied formal specifications of its semantics (in terms of process algebra [20]) can be used for computer-aided verification. This framework does not support runtime verification on production infrastructures.

The approach presented in [21] aims at dealing with environmental dynamism in service-based applications. The proposed modeling approach allows for partial definition of services in order to perform late (i.e., at runtime) incremental composition, when the execution context is discovered. This approach does not support formal verification of requirements at design-time.

4 Conclusion and Future Work

This paper describes an ongoing research activity on the application of formal methods to continuously support the development of microservices based cloud applications. The approach gets solid foundations from well-established formal methods and connects them to microservices based process flows. In particular, we make use of TB nets formalism to support design- to run-time verification of cloud applications built on top of Conductor.

The design-time phase aims at coping with continuously evolving specifications by keeping automatically updated the formal specification. The RV phase provides a way to support integration testing activity in a formal setting by means of a single software tool, although each service is independent and potentially implemented using different programming languages and technologies.

We are going to validate the RV module with a realistic case study deployed on a cloud platform. In addition, we are interested in expanding on this work in different directions. We want to extend the RV module in order to support online model-based testing with different *scenario control* techniques [12]. Moreover, we aim at expanding the translation to stochastic PNs (supporting probabilistic model checking) to deal with the intrinsic uncertainty of the environment.

References

1. N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: yesterday, today, and tomorrow,” in *PAUSE: Present and Ulterior Software Engineering*, B. Meyer and M. Mazzara, Eds. Springer, 2017, to appear. [Online]. Available: https://arxiv.org/pdf/1606.04036.pdf

2. "Microservices: a definition of this new architectural term," https://martinfowler.com/articles/microservices.html, last visited: June 2017.
3. T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.
4. "Netflix, Inc." https://www.netflix.com/, last visited: June 2017.
5. "Conductor," https://netflix.github.io/conductor/, last visited: June 2017.
6. C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè, "A unified high-level Petri net formalism for time-critical systems," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 160–172, February 1991.
7. M. Camilli, A. Gargantini, and P. Scandurra, "Specifying and verifying real-time self-adaptive systems," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2015, pp. 303–313.
8. M. Camilli, "Petri nets state space analysis in the cloud," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1638–1640.
9. M. Camilli, A. Gargantini, P. Scandurra, and C. Bellettini, "Event-based runtime verification of temporal properties using time basic Petri nets," in *NASA Formal Methods: 9th Int. Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, ser. LNCS, C. Barrett, M. Davies, and T. Kahsai, Eds. Cham: Springer Int. Publishing, 2017, vol. 10227, pp. 115–130.
10. "Uber Technologies, Inc." https://www.uber.com/, last visited: June 2017.
11. R. Alur, C. Courcoubetis, and D. Dill, "Model-checking in dense real-time," *Inf. Comput.*, vol. 104, no. 1, pp. 2–34, May 1993.
12. M. Veanes, C. Campbell, W. Schulte, and N. Tillmann, "Online testing with model programs," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 273–282, Sep. 2005.
13. "Conductor2Pn," https://bitbucket.org/seresearch_unimi/conductor2pn, last visited: June 2017.
14. M. Camilli, C. Bellettini, L. Capra, and M. Monga, "CTL model checking in the cloud using mapreduce," in *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Sept 2014, pp. 333–340.
15. H. Liang, J. S. Dong, J. Sun, and W. E. Wong, "Software monitoring through formal specification animation," *Innovations in Systems and Software Engineering*, vol. 5, no. 4, pp. 231–241, 2009.
16. C. Ramchandani, "Analysis of asynchronous concurrent systems by timed Petri nets," Cambridge, MA, USA, Tech. Rep., 1974.
17. W. J. Lee, S. D. Cha, and Y. R. Kwon, "Integration and analysis of use cases using modular Petri nets in requirements engineering," *IEEE Trans. Softw. Eng.*, vol. 24, no. 12, pp. 1115–1130, Dec. 1998.
18. S. Hinz, K. Schmidt, and C. Stahl, "Transforming BPEL to Petri nets," in *Proc. of the 3rd Int. Conf. on Business Process Management*, ser. BPM'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 220–235.
19. F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro, "JOLIE: a java orchestration language interpreter engine," *Electr. Notes Theor. Comput. Sci.*, vol. 181, pp. 19–33, 2007.
20. W. Fokkink, *Introduction to Process Algebra*, 1st ed., W. Brauer, G. Rozenberg, and A. Salomaa, Eds. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2000.
21. A. Bucchiarone, M. De Sanctis, and M. Pistore, *Domain Objects for Dynamic and Incremental Service Composition.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 62–80.