Continuous Formal Verification of Microservice-Based Process Flows

Chapter · September 2020

DOI: 10.1007/978-3-030-59155-7_31

1 author:

Matteo Camilli
Free University of Bozen-Bolzano
35 PUBLICATIONS **177** CITATIONS

SEE PROFILE

Continuous Formal Verification of Microservice-based Process Flows

Matteo Camilli[0000−0003−2491−5267]

Faculty of Computer Science
Free University of Bozen-Bolzano, Italy
matteo.camilli@unibz.it

Abstract. The microservice architectural style is often used to implement modern cloud, IoT, and large-scale distributed applications. Here software development processes are characterized by short incremental iterations, where several updates and new functionalities are continuously integrated many times a day in a agile fashion. Such a paradigm shift calls for new formal approaches to systematic (design-time and runtime) verification. This paper introduces a formal framework to apply continuous verification of microservice based applications built on top of Conductor, i.e., an open source orchestration engine of microservices workflows in use at Netflix, Inc. for their production environment. Our proposal adopts a model-driven paradigm and it leverages solid foundation from Petri nets to specify and verify the behavior of time-dependent workflows. This paper describes our approach, the current implementation, and evaluation activity conducted on a taxi-hailing application example.

Keywords: Microservices · Petri Nets · Formal Verification · DevOps.

1 Introduction

Microservices [11] represents an upward trending architectural style of modern cloud, IoT, or more in general advanced large-scale distributed applications. Even though fundamental principles of microservices are not novel or innovative[1], the migration towards microservices is still a sensitive matter nowadays. In fact, several leading companies applied huge reengeneering activities to adopt this paradigm. As a notable example, Netflix, Inc. [26] moved successfully from a monolithic architectural style to a microservices-based architecture in order to stream multimedia contents to an unprecedented amount of users every day. The adopted architecture builds upon the Netflix Conductor engine [10], an open source framework designed by Netflix Inc. and used daily in their production environment. Conductor allows the creation of arbitrary complex workflows in which individual tasks are implemented by microservices. The workflow *blueprint* (i.e., a high level description of the control and data flow) is defined using a JSON

[1] They are comparable to those of service-oriented computing [13] and we can find their roots in the design principles of Unix [15].

based DSL and includes a set of *worker* tasks (i.e., pieces of functionality) running on compute nodes and *system* tasks (i.e., the glue composing the workflow) executed by Conductor. Here, verification activities or even testing can be challenging. In fact, continuous changes in rapidly evolving settings potentially require continuous verification methods where artifacts must be constantly recreated or modified. Moreover, the polyglot nature associated with microservices potentially requires multiple verification/testing tools because of different programming languages and runtime environments. To deal with these issues, we introduce a formal framework to support continuous verification of microservice workflows built on top of Conductor. Our approach extends our previus work introduced in [6] and it adopts a model-driven [21] paradigm that pushes the usage of formal models through the development as well as operation phases. We foster the integration of the approach in modern software development practices, such as DevOps [12], in order to adopt formal methods in agile, continuous delivery, and automation setting. To achieve this goal, we decrease the cost of producing a formal specification by means of an automated model to model transformation technique. Namely, we mechanically obtain a Time Basic Petri Net [16] (or simply TB net) formal model from the Conductor blueprint. TB nets represents a time-extension of Petri nets (PNs) provided with a clear formal semantics, traditionally considered as effective formal specification of distributed systems with time constraints. The TB nets formalism is supported by powerful off-the-shelf software tools covering both modeling and verification phases [4]. Generated models can be used to perform computer aided verification activities such as model checking by means of well-known techniques. Once the model has been verified, it can be used at runtime, after the deployment of a release build, to monitor and verify the behavior of the target application with respect to its formal specification. Both the model transformation process and the runtime verification technique are currently implemented as part of a open source software toolchain. We used our continuous verification framework to verify both behavioral and temporal properties of a microservice-based taxi-hailing application built upon the Conductor engine. Results obtained from this preliminary validation activity are presented and discussed.

The paper is organized as follows. In Sect. 2 we give a preliminary high-level overview of our continuous verification approach. In Sect. 3, we introduce background notions to make this paper self-contained and our taxi-hailing microservices-based application running example. In Sect. 4 we provide a detailed description of our framework. In Sect. 5 we discuss the evaluation activity conducted on the taxi-hailing example. In Sect. 6 we present related work. Finally, we draw our conclusions in Sect. 7.

2 Overview of the Approach

Fig. 1 shows an high-level overview on the main phases and how they can be integrated into modern software development practices, such as a DevOps setting [12]. The guidelines of DevOps define a handshake between development

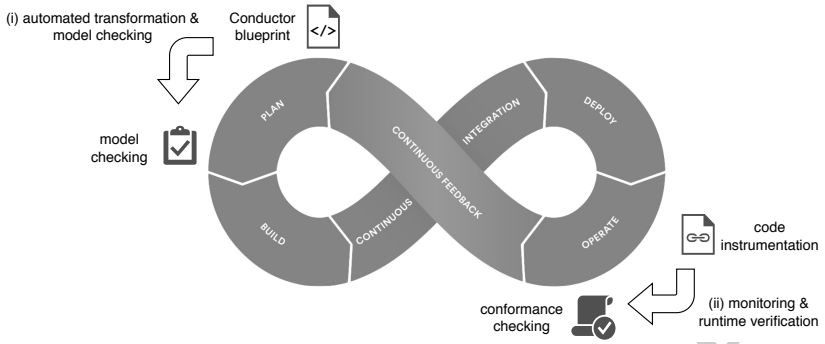


Fig. 1: Overview of our approach integrated into a DevOps setting.

and operations that forces a shift in mindset, better collaboration, and tighter integration. Following this trend that emphasizes fading boundaries between design-time and runtime phases, we introduce a continuous formal verification approach based on the iteration of the two phases: (*i*) Model Transformation & Model checking; and (*ii*) Monitoring & Runtime Verification. Although the approach is general, here we focus on microservices and the Conductor engine. Some of the peculiar characteristics of Conductor will be leveraged to introduce the rationale and put into place in a natural way some major technical details of our approach.

(i) Model Transofrmation & Model checking. In this phase we automatically generate the formal specification of the target application by transforming the Conductor blueprint into a TB nets model which describes both the system under development. The goal of this automatic transformation process is to aid the creation of formal models in rapidly evolving conditions. In fact, every change made to the workflow blueprint during development can be automatically reflected to the formal TB net specification, lowering the cost of keeping it consistent along the software lifecycle. We leverage TB net places to represent the *status* of a service (i.e., scheduled, in progress, timed out, or failed) and transitions to represent both service primitives and events coming from the surrounding environment. We leverage time modeling capability of TB nets to specify temporal constraints on scheduling/execution of tasks composing the overall applications. We defined a complete formal semantics of Conductor-based workflows, meaning that we cover all the available language constructs to define a Conductor blueprint. The final model is given by the composition of TB net transformation patterns derived from microservice and execution flow constructs. Such a model can be used to perform formal verification activities, such as interactive simulation (e.g., using token game) and model checking, with the aid of existing off-the-shelf software tools. Our current implementation focuses on the verification of Time Computation Tree Logic (TCTL) formulas [1] to verify *deadlock*/*livelock* freedom, *invariant*, *safety*, *liveness* and *bounded response-time* properties.

(ii) Monitoring & Runtime Verification. A model (re)generated in the previous step can be used to perform runtime verification upon the production infrastructure (i.e., the CONDUCTOR engine usually running on a cloud platform). The objective here is to run and monitor the execution of the target application in order to check conformance with respect to its own formal specification, thus enabling faster feedback which lays the foundation of every high performing DEVOPS team. The adopted runtime verification technique supplies the ability to map methods of interests (called *action method*s) to specific components of the specification (i.e., TB net transitions). During operations, we perform a monitoring activity through the co-execution of the target application and its formal specification (triggered by observable events). The monitor continuously evaluates the conformance of the execution (timed) trace with respect to the TB net model and it produces a (on-the-fly) report about both functional and temporal conformance failures. The report can be used in turn by developers to extract insights on the failing components. Our current implementation makes use of an JAVA monitoring engine which leverages the ASPECTJ framework to instrument the execution of the CONDUCTOR orchestrator.

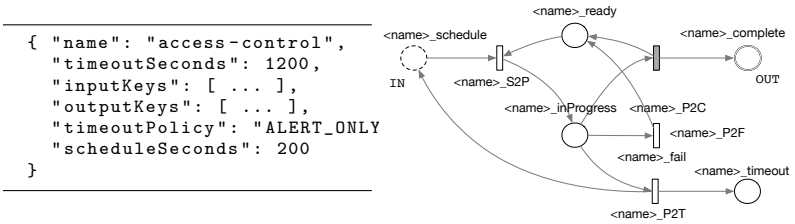
It is worth noting that in a continuous integration and delivery pipeline, teams usually have different deployment environments including testing and production. Our RV approach is intended to be integrated in each one of these environments. In fact, common practices in such as load/stress testing can be used to assess to what extent synthetically generated workload intensities affect the ability to verify formal requirements. Furthermore, RV in production can constantly monitor the target system and provide insights on occurring conformance failures.

3 Preliminaries

3.1 Time Basic Petri nets

TB nets represent an effective formal specification of concurrent (distributed) time-dependent systems. Time constraints are introduced as linear functions associated with each transition representing possible firing instants computed since transition's enabling. Tokens are atomically produced by firing transitions and they are timestamped along with time values ranging over $\mathbb{R}_{\geq 0}$. TB nets support a mixed time semantics, i.e., both *urgent* and *non-urgent* transitions can be used to define mandatory and optional events, respectively.

The structure of a TB net extends the P/T net one (P, T, F), where P is a finite set of places, T is a finite set of transitions such that $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (or flows) connecting places to transitions and transitions to places. Let $v \in P \cup T$: ${}^\bullet v$, $v^\bullet$ denote the backward and forward adjacent sets of v according to F, respectively, also called pre/post-sets of v. A timestamp *binding* of $t \in T$ is a map $b_t : {}^\bullet t \rightarrow Bag(\mathbb{R}_{\geq 0})$. Moreover, each transition t is associated with a *time function* f_t which maps a binding b_t to a (possibly empty) set of $\mathbb{R}_{\geq 0}$ values, denoted by $f_t(b_t)$. f_t is formally defined as a pair of linear functions $[l_t, u_t]$, denoting parametric interval bounds.



(a) JSON worker task

(b) TB net transformation pattern

Initial marking: $\langle$name$\rangle$_ready$\{T_A\}$

Transition	**Time function**
$\langle$name$\rangle$_S2P	$[\tau_e, \tau_e + \langle$scheduleSeconds$\rangle]$
$\langle$name$\rangle$_P2C	$[\tau_e, \tau_e + \infty]$
$\langle$name$\rangle$_fail	$[\tau_e, \tau_e + \infty]$
$\langle$name$\rangle$_P2T	$[\tau_e + \langle$timeoutSeconds$\rangle, \tau_e + \langle$timeoutSeconds$\rangle]$

Fig. 2: Transformation pattern of a `ALERT_ONLY` timeout-policy worker. Non-urgent transitions are depicted in gray.

A *marking* (or state) is a mapping $m : P \to Bag(\mathbb{R}_{\geq 0})$, where $Bag(X)$ represents all possible multisets over X. According to the *non-urgent* (or *weak*) semantics, t can fire at any instant $\tau \in f_t(b_t)$. The *urgent* (or *strong*) interpretation states that t must fire at any $\tau \in f_t(b_t)$, unless disabled by the firing of any conflicting transitions before the latest firing time of t. Given a binding b_t, a pair (b_t, τ), with $\tau \in f_t(b_t)$, represents a firing instance of t. The firing instance produces a new reachable marking by applying the traditional PN *firing rules*, but producing tokens timestamped with τ.

Fig. 2b shows a TB net example that models the lifecycle of a single microservice. A single token with timestamp $T_0 = 0$ in place $\langle$`name`$\rangle$`_schedule` represents the microservice $\langle$`name`$\rangle$ in scheduled state (at time 0). In this marking, the transition $\langle$`name`$\rangle$`_S2P` is the only one enabled to fire by the binding: $\{\langle$`name`$\rangle$`_schedule` $\to \{1 \cdot T_0\}, \langle$`name`$\rangle$`_ready` $\to \{1 \cdot T_A\}\}$. The variable T_A represents a special timestamp (i.e., anonymous timestamp) whose time value does not influence the evolution of the system. Possible firing time instants are obtained by evaluating the bounds of $f_{\langle\texttt{name}\rangle\texttt{\_S2P}}$: $[\tau_e, \tau_e + 200]$, where τ_e is the transition's enabling time (the value 0 in this case). Given a valid timestamp value $\tau \in [0, 200]$ (e.g., the value 150), according to the firing rules, we get a new marking with a new token $T_0 = 150$ in place $\langle$`name`$\rangle$`_inProgress` (i.e., the execution of the microservice starts from time 150). In this new marking, three transitions are concurrently enabled to fire: the *non-urgent* $\langle$`name`$\rangle$`_P2C` in the time interval $[0, \infty]$; and the two *urgent* transitions $\langle$`name`$\rangle$`_fail` and $\langle$`name`$\rangle$`_P2T` in the time interval $[1200, 1200]$. This configuration shows that the service can either complete the execution, fail or enter a timeout state. In the latter case, the system increments a counter (by producing a token into place $\langle$`name`$\rangle$`_timeout`) and then schedules again the service. Whenever a final state is entered (i.e., either

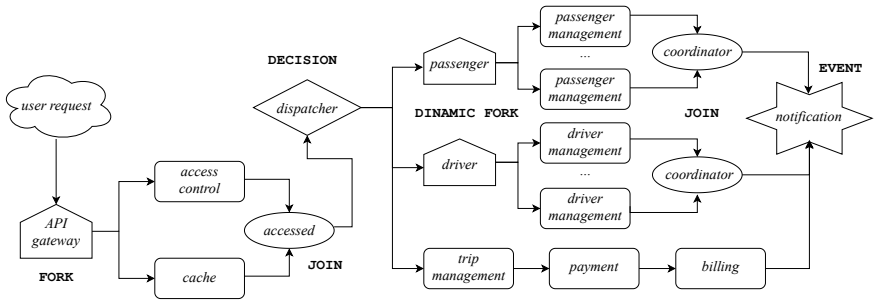


Fig. 3: High-level schema of the taxi-hailing blueprint.

the place ⟨`name`⟩`_complete` or the place ⟨`name`⟩`_fail` is marked), the microservice returns in ready state to serve new requests.

Formally, a marking m_n is *reachable* from m_0 iff. there exists a *path* σ (sequence of firing instances and markings) such that:

$$\sigma = m_0 \xrightarrow{(b_{t_0}, \tau_0)} m_1 \xrightarrow{(b_{t_1}, \tau_1)} m_2, \ldots, m_{n-1} \xrightarrow{(b_{t_{n-1}}, \tau_{n-1})} m_n$$

The transitions associated with the enabled bindings in m are called *enabled transitions* and they are denoted by $enab(m)$.

By using consolidated analysis techniques it is possible to construct a finite symbolic state space of a TB net model, called its *Time Reachability Graph* (*TRG*) [7,5]. The *TRG* construction is fully automated and it relies on a *symbolic state* notion: each reachable state is a pair: $S = (M, C)$, where M (symbolic marking) maps places into multisets of timestamps and C (constraint) is a logical predicate formed by linear inequalities defining time relations between timestamps. Given the *TRG* structure, model checking algorithms can be applied to verify the correctness of the system against requirements expressed as Time Computation Tree Logic (TCTL) properties [1,3]. The model checking technique is fully automated by the GRAPHGEN software tool.

3.2 A Running Example

We introduce here a small taxi-hailing workflow example used to put into place major concepts. Fig. 3 shows an high-level view of the provided pieces of functionality. This schema follows the notation introduced in [10] and shows both services and their relations in a CONDUCTOR workflow. Each microservice (rectangle) implements an isolated function (e.g., access control, trip management, payment, etc.) and is deployed independently, usually into cloud virtual machines or Docker containers [24]. Microservices expose REST APIs consumed by other services. For instance, *passenger management* uses the *notification* service to notify a passenger about an available driver. The *API gateway* exposes a public API used by mobile clients or web UIs. Other shapes represent control and data flow primitives (e.g, `EVENT`, `FORK`, and `JOIN`) executed by the CONDUCTOR

Table 1: Taxi-hailing workflow requirements

label	description	property-type	CTL formula
R_1	The payment service cannot reach an inconsistent state where both in progress and timeout status coexist	safety	$\neg EF(\texttt{payment\_inProgress} > 0 \wedge \texttt{payment\_timeout} > 0)$
R_2	Whenever a a user request has been handled correctly, then a task among Driver, Passenger and Trip management is executed	liveness	$AG(\texttt{accessControl\_complete} > 0 \wedge \texttt{cache\_complete} > 0 \rightarrow AF(\texttt{Passenger\_schedule} > 0 \vee \texttt{Driver\_schedule} > 0 \vee \texttt{TripManagement\_schedule} > 0))$
R_3	Whenever a payment task is scheduled for execution, it is possible to complete the billing process in 2.4 seconds	bounded response-time	$AG(\texttt{payment\_schedule} > 0 \rightarrow EF_{\leq 2400}(\texttt{billing\_complete} > 0))$

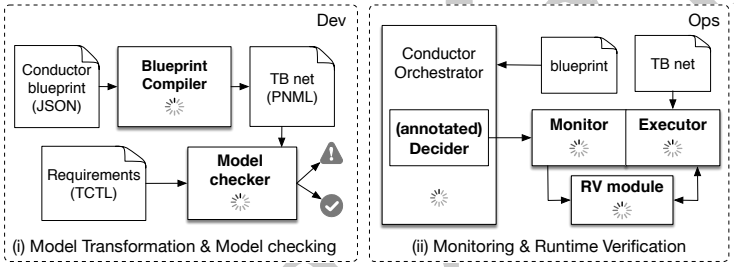


Fig. 4: Toolchain supporting our approach.

orchestrator. As an example, the `DECISION` *dispatcher* allows to choose between alternative flows depending on the request type. The *passenger* and the *driver* components use the `DYNAMIC FORK` primitive to send user requests to different (replicated) services.

Finally, let us assume that the taxi-hailing workflow must satisfy the requirements reported in Table 1. Requirements are formally expressed as TCTL properties to verify them upon the TB net specification.

4 Continuous Formal Verification

Fig. 4 shows the major components of the toolchain and their existing relations. As anticipated in Sect. 2, the approach builds upon a model-driven iterative paradigm aiming at providing support to both development and operation phases in a formal fashion.

4.1 Model Transformation & Model Checking

The first step is a fully automated model-to-model transformation carried out by the *Blueprint Compiler* module as shown in Fig. 4. Our technique follows

the approach introduced in [6] and it provides transformation capability for each construct of the Conductor (JSON-based) specification language. The overall process is guided by the identification of transformation patterns of each individual microservice (worker task) and each workflow primitive (system task). Patterns have input/output elements to compose them each other. The final TB net model is the result of the composition (i.e., the union by connecting input/output elements) of different transformation patterns of the corresponding microservices and primitives.

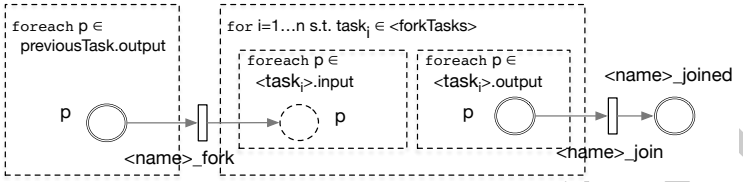
Worker Tasks – We use TB net places to represent the state of a task, while TB net transitions represent task primitives. Temporal functions associated with transitions are used to specify time concerns of scheduling and execution. Fig. 2a shows the definition of a worker task using the Conductor language. The listing contains a JSON object with a number of control parameters used to tell the orchestrator how to manage the microservice lifecycle. The `scheduleSeconds` parameter sets an upper bound to the scheduling time of each instance of the worker task. The `timeoutSeconds` sets instead an upper bound to the execution time. Thus, if the `access-control` does not complete in 1200 milliseconds, the Conductor orchestrator must kill the execution and alert the system (by incrementing a timeout counter) because of the `ALERT_ONLY` timeout-policy. Fig. 2b shows the corresponding TB net transformation pattern. This pattern must be instantiated by replacing each ⟨`parameter`⟩ with the corresponding value in the JSON object. Dashed line shapes represent the input elements (e.g., `access-control_schedule`). Double line shapes represent output elements (e.g., `access-control_complete`). Three different types of timeout-policy exist: `ALERT_ONLY`, `TIMEOUT_WF` (i.e., put the entire workflow in timeout state), `RETRY` (i.e., reschedule the worker task a fixed number of times). The Conductor engine handles the execution of worker tasks depending on values assigned to timeout-policy and retry-logic. As a consequence, these control parameters are used to identify the right transformation pattern.

A detailed description of the behavior of all possible types of worker is outside the scope of this paper and can be found in [10]. We let the reader refer to [6] for a comprehensive discussion about (TB net) formalization of worker tasks.

System Tasks – In addition to a sequence of worker tasks, the Conductor blueprint declares a number system tasks representing synchronization primitives. In the following we provide the reader with a representative example of transformation used in our taxi-hailing application, i.e., the `FORK_JOIN` *API gateway*. Such a primitive is used to schedule a parallel set of tasks specified in the control parameter `forkTasks` by a list of task sequences. Fig. 5a shows the listing used to define this task in our running example. Here the two parallel sequences contain a single worker task: the `access-control` and the `cache`, respectively. This means that upon a user request, performed though the API, the orchestrator triggers a parallel scheduling/execution of both microservices. The control parameter `joinTasks` contains the lists of tasks whose completions determines the end of the fork execution. If both access control and the cache services succeed, the system replies back to the client through a notification event.

```
{ "name": "api-gateway",
  "type": "FORK_JOIN",
  "forkTasks": [[{"taskReferenceName": "access-control", "type": "SIMPLE"}],
                [{"taskReferenceName": "cache", "type": "SIMPLE" }] ],
  "joinOn": ["access-control", "cache" ],
  "forkSeconds": 250,
  "joinSeconds": 250
}
```

(a) JSON `FORK_JOIN` task



(b) TB net transformation pattern

Transition	Time function
⟨name⟩_fork	$[\tau_e, \tau_e + \langle \text{forkSeconds} \rangle]$
⟨name⟩_join	$[\tau_e, \tau_e + \langle \text{joinSeconds} \rangle]$

Fig. 5: Transformation pattern of a `FORK_JOIN` system task.

Otherwise, if the required information is not cached, the process continues the execution with the `DECISION` (req. type decision) system task. Fig. 5b show the associated transformation pattern composed of elementary TB net structural elements and *macro substitution*s, delimited by dashed boxes. A `for` macro substitution is a construct used to repeat the inner elements depending on the attached annotation. As an example, the ⟨`name`⟩`_fork` transition represents the starting point of the fork tasks and must be connected to all the input elements ($p \in \texttt{task}_i\texttt{input}$) of all the tasks declared in the listing ($\texttt{task}_i \in \texttt{forkTasks}$). The parameters `forkSeconds` and `joinSeconds` define the maximum time (milliseconds) required by the fork and the join operations, respectively. These values are used to instantiate the time functions of the pattern as shown in Fig. 5b.

Composition and Model Checking – The overall transformation process, executed by the blueprint compiler, reduces to the application of two steps in sequence: transformation of each worker task; and then worker composition, following the definition of transformation patterns associated with the declared system tasks. The result of this process is a TB net model formally specifying the behavior of the overall workflow. The formalization enables the usage of verification techniques to assess design-time requirements satisfaction. In our current approach we use interactive simulation (token game) to support validation, and TCTL model checking to support formal verification. Important properties that can be checked include deadlock/livelock freedom of the workflow, invariant, safety, liveness and bounded response-time properties. For instance, the TCTL

properties R_1, R_2 and R_3 reported in Table 1 have been verified on the taxi-hailing workflow specification.

4.2 Monitoring & Runtime Verification

The generated TB net model is kept alive during operations in order to monitor the target workflow. We use the RV technique to verify conformance of behavioral and temporal aspects by first extracting a *timed-trace* of occurring events, from the running workflow, and then verifying whether it corresponds to a feasible *execution path* in the TB net model. A description of the RV approach follows.

Given a workflow ω, we denote a timed-trace π_i as a sequence of *observable event*s $\pi_i = \{e_1, ..., e_n\}$, where each event e_k represents the execution of a task (either worker or system) that causes ω to change its global state. An observable event e_k is formally identified by the pair $\langle id(e_k), time(e_k)\rangle$, where *id* and *time* map e_k to a identifier (sequence of characters) and a timestamp (in $\mathbb{R}_{\geq 0}$), respectively. An example of timed-trace, extracted from the taxi-hailing workflow, follows.

$$\begin{aligned}\pi_i = \; & e_0 : \langle \texttt{api-gateway\_fork}, 450\rangle, e_1 : \langle \texttt{access-control\_S2P}, 622\rangle, \\ & e_2 : \langle \texttt{cache\_S2P}, 630\rangle, e_3 : \langle \texttt{access-control\_P2C}, 1550\rangle, \dots\end{aligned} \tag{1}$$

Intuitively, the *Monitor* component is in charge of extracting the timed trace π_i from the execution of ω. The *Executor* component incrementally builds the execution path σ_i from the TB net model depending on the occurring observable events. The *conformance relation* is checked by the *RV* module on-the-fly during the co-execution of the workflow and the model by performing a pairwise comparison of occurring events in ω and firing transitions in the mdoel.

Formally, there exists a *conformance relation* between π_i and σ_i, iff. for each $e_k \in \pi_i$, there exists $m_k \in \sigma_i$ such that:

$$t_k \equiv id(e_k) \wedge time(e_k) \in [l_t(b_{t_k}), u_t(b_{t_k})] \tag{2}$$

To verify such a relation, for each occurring event $e_k \in \pi_i$, the *RV* module verifies that the model transition t named $id(e_k)$ is enabled to fire from the current marking $m \in \sigma_i$. The observed timestamp must conform to possible firing times of t. If this condition holds, the *Executor* component updates σ_i creating a new reachable marking with the proper timestamp.

The workflow execution is made observable by using AspectJ instrumentation of the Conductor Decider source code. In fact, the Decider contains a number of callback used to handle workflow events, like task scheduling, completion, and failure. The annotation allows the execution of the callback methods to be intercepted by AspectJ. Thus, callbacks generate observable events for the *Monitor*. The *Monitor* enqueues the event e_k into the trace π_i, depending on the workflow id. The *RV* module computes $id(e_k)$ by concatenating the task reference name and status. Then, it retrieves $time(e_k)$ by sampling the time from the Java virtual machine. If the conformance relation does not hold, a conformance failure exception is thrown. The exception shows information about the (timestamped) event that generated the exception, along with the set of enabled

Table 2: Taxi-hailing structure.

Conductor blueprint		TB net transformation		
#worker tasks	#system tasks	$\|P\|$	$\|T\|$	TRG #states
9	6	57	61	8854

Table 3: Design-time Verification.

Transformation		TRG building		model checking	
time (s)	space (KB)	time (s)	space (KB)	time (s)	space (KB)
2.13	80,304	210.12	273,166	(1) 0.19	(1) 7,741
				(2) 0.38	(2) 9,522
				(3) 0.22	(3) 7,011

bindings that represent in this context the expected events predicted by the model.

5 Experimental Validation

We validated the overall continuous verification approach by conducting a number of experiments using our taxi-hailing example both during development and operation phases. Experiments have been conducted on a machine equipped with a Intel Xeon E5-2630 at 2.30GHz CPU, 64GB of RAM, the Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-39-generic x86_64) operating system with a completely fair scheduler, and the Java HotSpot 1.8 64-Bit Server Virtual Machine using the Garbage-First (G1) collector. Here we briefly discuss some significant results and we refer the reader to our implementation[2] for the replicability of the experiments.

Design-time Verification – Data describing structural properties of the taxi-hailing blueprint and corresponding TB net transformation are reported in Table 2. This table reports the TB net model size in terms of number of places $|P|$ and number of transitions $|T|$ and the *TRG* size in terms of number of reachable states. Table 3 reports the execution time (in seconds) and the average memory consumption (in KBytes) of a number of operations required by verification at design-time. Data shows the most expensive operation is the *TRG* building, while both transformation and verification are orders of magnitude cheaper both in terms of execution time and memory consumption. The transformation process, in particular, is very efficient because its complexity strictly depends on the model size (i.e., the structural complexity of the blueprint and the TB net) which is a small structure with respect to the *TRG* size ($\sim 10^1$ vs $\sim 10^3$ in our taxi-hailing example).

Runtime Verification – Here we discuss experimental results to evaluate the overhead of the runtime verification module running along with the taxi-hailing Conductor workflow. Table 4 shows data extracted during runtime verification activities by varying the frequency, i.e., average number of *Monitor* invocations per second. The metrics used during the evaluation include the

[2] The main components of the toolchain are available as open source software at `https://github.com/SELab-unimi/conductor2pn` and `https://maharajaframework.bitbucket.io/`.

Table 4: Runtime Verification performance.

frequency (#invocations × s.)	AJO (μs)	AJO jitter (μs)	MIO (μs)	MIO jitter (μs)	DL (μs)	Memory (KB)
1	55.0	23.2	24.0	43.5	1488.6	2,113
2	52.5	20.6	23.6	46.4	1205.9	3,488
4	49.4	24.8	25.0	45.3	1022.7	5,055
8	52.8	27.6	28.1	47.4	755.8	10,066

amount of monitoring overhead added by the instrumentation, the overhead jitter, and the auxiliary memory usage.

The *monitoring overhead* is caused by two main factors: the AspectJ instrumentation (AJO) and the monitor invocation overhead (MIO). Table 4 reports the average value of this two variables (in μs) we observed during the execution of the instrumented Conductor orchestrator by varying the frequency value. The average AJO (i.e., due to the invocation of AspectJ advices) strictly depends on the byte code generated from the annotated Decider by using the AspectJ compiler. The order of magnitude of measured AJO values is approximately $10\mu s$. The MIO is caused by the amount of time required to enqueue an occurring event into the synchronized event buffer structure. We observed that both the MIO and the AJO have the same order of magnitude, however the average MIO is generally lower ($\sim$50% lower). Namely, the overhead introduced by AspectJ dominates the overall monitoring overhead. The monitor invocation frequency impacts on the average MIO. We observed a linear correlation between MIO and frequency values. The *overhead jitter* represents the deviance between the monitoring overhead values. Results show that AJO and the MIO jitter values have the same order of magnitude (i.e., $\sim 10\mu s$). While the AJO jitter strictly depends on the mechanics of AspectJ, the MIO jitter is governed by the *Monitor* status during the observation of events. We observed MIO bursts whenever observable events occur while the *Monitor* is suspended. In fact, in such a case, the MIO includes the time required by resuming the suspended monitoring thread before enqueuing a new event into the empty event buffer synchronized structure. The *detection Latency* value represents the time between an occurring event and the verdict (i.e., either conformance checked or conformance failure) computed by the *RV* module. A bounded detection latency (DL) allows for fast identification of conformance failures, thus making the operations team able to promptly react to degraded situations. During our experimentation we observed the following trend: the higher the frequency, the lower the DL. In fact, a low frequency implies very often an empty event buffer structure, thus increasing the overhead required by resuming a suspended thread. Overall we observed that the *RV* module is able to identify a conformance failure with a very small DL ($\sim$1 millisecond). The *memory overhead* is the additional space used by the Java virtual machine to run and runtime verification components. Table 4 shows negligible auxiliary memory values (few KBytes on average). We observed a linear correlation between memory overhead and monitor invocation frequency.

6 Related Work

The approach presented in this paper has been mainly influenced by different related works on formal specification and verification techniques of (micro)service-based systems. In particular, we leverage formal methods and integrate them into modern agile practices by following the approach envisioned in [17].

Although modeling formalisms such as timed-automata [2] or finite-state-machines [18] support the modeling of temporal or behavioral aspects, PNs-based approaches are generally more concise and scalable in the specification of concurrency and distribution [23]. Furthermore, aspects such as messaging, communication protocols, which are commonly used in distributed architectures, such as service oriented architectures and microservices, can be difficult to model with the language primitives of automata-based formalisms [20,23]. PNs represent common formal models of service-oriented architecture specified by means of the Business Process Execution Language for Web Services (BPEL) as described in [19]. However, BPEL transformation approaches cannot be directly applied in the context of microservices, where new emerging languages and frameworks, such as Conductor and Jolie [25], represent upward trending choices. Jolie is a microservices workflow interpreter engine equipped with a formal semantics in terms of process algebra [14] that can be used for computer-aided verification at design-time. Another recent line of research aims at leveraging the Event-B modeling language to define microservices architectural patterns [27]. The approach provides formal models of these patterns with the final goal of improving comprehension and enabling correct-by-construction mechanisms. Our runtime verification technique has been built upon the approach presented in [8], i.e., an event-based Runtime Verification (RV) technique for temporal properties of distributed systems leveraging TB nets as modeling formalism. As described in [8] the technique is supported by off-the-shelf tools that outperform comparative other representative state-of-the-art runtime verification Java software tools such as Java MaC [22], and Larva [9].

7 Conclusion

This paper describes an ongoing research activity on the application of formal methods to continuously support the development and operation phases of microservices-based workflows. The approach uses a model-driven paradigm and exploits solid foundation from well-established formal methods. Namely, we use the expressiveness of TB nets to support continuous verification of Conductor workflows. Model transformation and design-time verification performed during development aims at coping with continuously evolving specifications by keeping (verified) artifacts automatically updated. Runtime verification provides a way to support operation phases by monitoring and checking conformance of the target application with respect to its own formal specification in order to enable fast feedback and support high performing DevOps teams. The major components of our current toolchain have been released as open source software to encourage replication of experiments.

We are currently in the process of extending the RV technique to support (on-the-fly) model-based testing along with different scenario control techniques. We also want to expand the transformation capability by adding stochastic modeling of the intrinsic uncertain aspects of the surrounding environment.

References

1. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: [1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science. pp. 414–425 (Jun 1990). https://doi.org/10.1109/LICS.1990.113766
2. Bengtsson, J., Yi, W.: Timed Automata: Semantics, Algorithms and Tools, pp. 87–124. Springer Berlin Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_3
3. Camilli, M., Bellettini, C., Capra, L., Monga, M.: CTL model checking in the cloud using mapreduce. In: 2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. pp. 333–340 (Sept 2014). https://doi.org/10.1109/SYNASC.2014.52
4. Camilli, M., Gargantini, A., Scandurra, P.: Specifying and verifying real-time self-adaptive systems. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). pp. 303–313 (Nov 2015). https://doi.org/10.1109/ISSRE.2015.7381823
5. Camilli, M.: Petri nets state space analysis in the cloud. In: Proceedings of the 34th International Conference on Software Engineering. pp. 1638–1640. ICSE '12, IEEE Press, Piscataway, NJ, USA (2012)
6. Camilli, M., Bellettini, C., Capra, L., Monga, M.: A formal framework for specifying and verifying microservices based process flows. In: Cerone, A., Roveri, M. (eds.) Software Engineering and Formal Methods. pp. 187–202. Springer International Publishing, Cham (2018)
7. Camilli, M., Gargantini, A., Scandurra, P.: Zone-based formal specification and timing analysis of real-time self-adaptive systems. Science of Computer Programming **159**, 28 – 57 (2018). https://doi.org/https://doi.org/10.1016/j.scico.2018.03.002
8. Camilli, M., Gargantini, A., Scandurra, P., Bellettini, C.: Event-based runtime verification of temporal properties using time basic petri nets. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NASA Formal Methods. pp. 115–130. Springer International Publishing, Cham (2017)
9. Colombo, C., Pace, G.J., Schneider, G.: Formal Methods for Industrial Critical Systems: 13th Int. Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers, chap. Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties, pp. 135–149. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03240-0_13
10. Conductor, N.: Conductor documentation. `https://netflix.github.io/conductor/` (2019), online; accessed Sept. 2019
11. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: Yesterday, Today, and Tomorrow, pp. 195–216. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-67425-4_12
12. Ebert, C., Gallardo, G., Hernantes, J., Serrano, N.: Devops. IEEE Software **33**(3), 94–100 (May 2016). https://doi.org/10.1109/MS.2016.68

13. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River, NJ, USA (2005)
14. Fokkink, W.: Introduction to Process Algebra. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edn. (2000)
15. Fowler, M.: Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html (2019), online; accessed Sept. 2019
16. Ghezzi, C., Mandrioli, D., Morasca, S., Pezzè, M.: A unified high-level Petri net formalism for time-critical systems. IEEE Trans. Softw. Eng. **17**, 160–172 (February 1991). https://doi.org/10.1109/32.67597
17. Ghezzi, C.: Formal Methods and Agile Development: Towards a Happy Marriage, pp. 25–36. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-73897-0_2, https://doi.org/10.1007/978-3-319-73897-0_2
18. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Logic **1**(1), 77–111 (Jul 2000). https://doi.org/10.1145/343369.343384
19. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri Nets, pp. 220–235. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/11538394_15, http://dx.doi.org/10.1007/11538394_15
20. Iglesia, D.G.D.L., Weyns, D.: Mape-k formal templates to rigorously design behaviors for self-adaptive systems. ACM Trans. Auton. Adapt. Syst. **10**(3), 15:1–15:31 (Sep 2015). https://doi.org/10.1145/2724719
21. Kent, S.: Model driven engineering. In: Butler, M., Petre, L., Sere, K. (eds.) Integrated Formal Methods. pp. 286–298. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
22. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A run-time assurance approach for java programs. Form. Methods Syst. Des. **24**(2), 129–155 (Mar 2004). https://doi.org/10.1023/B:FORM.0000017719.43755.7c
23. Lee, W.J., Cha, S.D., Kwon, Y.R.: Integration and analysis of use cases using modular Petri nets in requirements engineering. IEEE Trans. Softw. Eng. **24**(12), 1115–1130 (Dec 1998)
24. Merkel, D.: Docker: Lightweight linux containers for consistent development and deployment. Linux J. **2014**(239) (Mar 2014), http://dl.acm.org/citation.cfm?id=2600239.2600241
25. Montesi, F., Guidi, C., Lucchi, R., Zavattaro, G.: JOLIE: a java orchestration language interpreter engine. Electr. Notes Theor. Comput. Sci. **181**, 19–33 (2007). https://doi.org/10.1016/j.entcs.2007.01.051, https://doi.org/10.1016/j.entcs.2007.01.051
26. Netflix, I.: The Netflix Service. https://www.netflix.com/ (2019), online; accessed Sept. 2019
27. Vergara, S., González, L., Ruggia, R.: Towards formalizing microservices architectural patterns with event-b. In: 2020 IEEE International Conference on Software Architecture Companion (ICSA-C). pp. 71–74 (2020)