A Formal Framework for Specifying and Verifying Microservices Based Process Flows

Matteo Camilli, Carlo Bellettini, Lorenzo Capra, Mattia Monga

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

Abstract. The microservices architectural style is changing the way in which software is perceived, conceived and designed. Thus, there is a call for techniques and tools supporting the problem of specifying and verifying communication behavior of microservice systems. We present a formal semantics based on Petri nets for microservices based process flows specified using the Conductor orchestration language: a JSON-based domain specific language designed by Netflix, Inc. We give a formal semantics in terms of a translation from Conductor specifications into Time Basic Petri net models, *i.e.,* Petri nets supporting the definition of temporal constraints. The Petri net model can be used for computer aided verification purposes by means of well-known techniques implemented by powerful, off-the-shelf model checking tools.

Keywords: Microservices, orchestration, formal methods, Petri nets, verification, time analysis

1 Introduction

One of the most successful mantras of the so called Unix' philosophy is: "Do one thing and do it well". In fact, the Unix' offspring is characterized by a highly component-oriented architecture, with many small and specialized black-boxes (like `grep`, `sort`, or `cut`) that people use everyday to assembly —often just by using the glue provided by the shell and its piping capabilities— higher level tasks. If Service Oriented Architectures (SOA) promise to bring black-box components to distributed systems, the current call for a *microservices* attitude aims at having small and specialized pieces of functionalities. According to some of the proposers of the term microservices, a single application should be built "as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API" [1, 2].

In order to establish cooperation among services, *orchestration* [3] has recently seen a renewed interest [4]. An orchestration engine is in charge of enacting a script (sometimes called the *blueprint* of the high level service) defining the high level control and data flows. Thus, in order to make the composite service predictable, it is important to develop the ability of reasoning at this higher level. Since orchestration languages have a very simple structure and the number

of components is usually small (for example, Netflix declares their "workflows" are made, on average, of six tasks, with the largest composed by 48 [5]), this seems a very good opportunity to apply formal methods, whose major weakness is often their scalability in front of real world applications complexity.

In this paper we propose to use Time Basic Petri nets [6], *i.e.,* a particular extension of Petri nets supporting the definition of temporal constraints, to analyze the properties of microservice-oriented applications orchestrated by the Netflix CONDUCTOR engine [4], an open source framework designed by Netflix Inc. and used daily in their production environment [5]. The CONDUCTOR 'Orchestrator' is driven by a workflow script, written in a JSON-based domain specific language. The Orchestrator tracks and manages workflows and it has the ability to pause, resume and restart the microservice tasks. We defined a formal semantics for the workflow language in which microservices are black-box described by Petri net modules. The formal semantics is supported by a Java tool CONDUCTOR2PN that translates CONDUCTOR specifications into a Time Basic Petri net model, which can be exploited for computer aided verification purposes by means of well-known techniques implemented by powerful, off-the-shelf model checking tools.

The paper is organized as follows: in Sect. 2 we recall some background notions on Time Basic Petri nets, in Sect. 3 we present a running example, in Sect. 4 we describe the semantics we defined, in Sect. 5 we apply the given semantics to verify some properties of the running example, in Sect. 6 we discuss some related work, and in Sect. 7 we presents our conclusions and future work.

2 Background

Time Basic Petri nets — Time Basic Petri nets (TB) nets are Petri nets (PNs or P/T nets) [7], where system 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 have timestamps with values ranging over $\mathbb{R}_{\geq 0}$. This modeling formalism represents an effective formal specification of time-dependent systems. It supports a mixed time semantics, *i.e.,* both *urgent* and *non-urgent* transitions can be used to define mandatory and optional events, respectively. TB nets are also nicely supported by powerful open source software tools considering both modeling and verification aspects [8, 9]. Although other modeling formalisms such as timed-automata [10] or finite-state-machines [11] support the modeling of temporal or behavioral aspects, PNs-based approaches can be more concise and scalable [12]. 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 [12, 13].

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

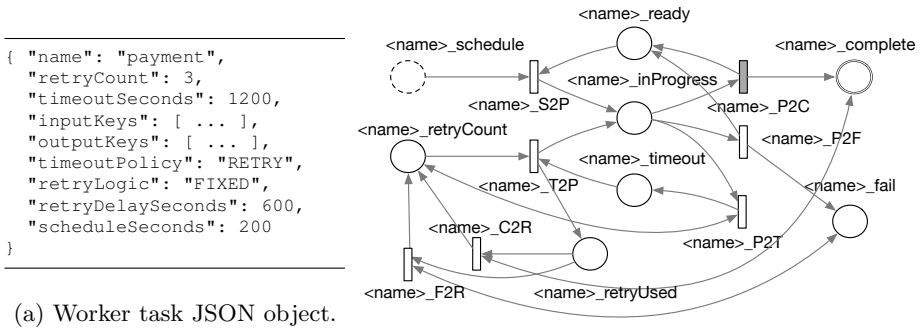
and transitions to places. Let $v \in P \cup T$: ${}^\bullet v$, $v^\bullet$ denote the backward and forward adjacent sets of v according to F, respectively, also called pre/post-sets of v. A timestamp *binding* of $t \in T$ is a map $b_t : {}^\bullet t \to Bag(\mathbb{R}_{\geq 0})$. Moreover, each transition 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 *marking* (or state) is a mapping $m : P \to Bag(\mathbb{R}_{\geq 0})$, where $Bag(X)$ represents the set of 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 an instant $\tau \in f_t(b_t)$), unless it is 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), s.t. $\tau \in f_t(b_t)$, is said a firing instance of t. The firing instance (b_t, τ) produces a new reachable marking m' by applying the following *firing rules*:

- $\forall p \in {}^\bullet t \setminus t^\bullet \; m'(p) = m(p) - b_t(p)$
- $\forall p \in t^\bullet \setminus {}^\bullet t \; m'(p) = m(p) + \{\tau\}$
- $\forall p \in t^\bullet \cap {}^\bullet t \; m'(p) = m(p) - b_t(p) + \{\tau\}$
- for all remaining places, $m'(p) = m(p)$

Figure 1b shows a TB net that models the lifecycle of a single microservice. We use it in the following to illustrate the background concepts. A single token with timestamp $T_0 = 0$ in place ⟨name⟩_schedule represents that the microservice ⟨name⟩ has been scheduled at time 0. In this marking, the transition ⟨name⟩_S2P is the only one enabled to fire by the following binding: $\{$⟨name⟩_schedule $\to \{1 \cdot T_0\}$, ⟨name⟩_ready $\to \{1 \cdot T_A\}\}$. Possible firing time instants are obtained by evaluating the parametric 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 ⟨name⟩_inProgress (*i.e.,* the execution of the microservice starts from time 150). In this new marking, two transitions are concurrently enabled to fire: the *non-urgent* ⟨name⟩_P2C in the time interval $[0, \infty]$, and the *urgent* ⟨name⟩_P2T in the time interval $[1200, 1200]$. Thus, the service can either complete the execution or enter a timeout state. In the latter case, the system retries to execute the service a fixed number of times, depending on the number of tokens in place ⟨name⟩_retryCount. Whenever the task is timed out (*i.e.,* the place ⟨name⟩_timeout is marked) and the retryCount limit has been reached (*i.e.,* the place ⟨name⟩_retryCount is empty), the task enters a failure state (*i.e.,* the place ⟨name⟩_fail is marked by the firing of ⟨name⟩_P2F). The symbol ε in $f_{\langle \texttt{name} \rangle \texttt{\_P2F}}$ represents an infinitesimal delay used to set a precedence between the conflicting transitions ⟨name⟩_P2T and ⟨name⟩_P2F. Whenever a final state is entered (*i.e.,* either the place ⟨name⟩_complete or the place ⟨name⟩_fail is marked), the transitions ⟨name⟩_C2R and ⟨name⟩_F2R restart the service by removing all the tokens from ⟨name⟩_retryUsed and refilling ⟨name⟩_retryCount.

Time Reachability Graph — By using consolidated analysis techniques it is possible to construct a finite symbolic state space of a TB net model, called



(a) Worker task JSON object.

(b) Translation pattern.

Initial marking: $\langle$name$\rangle$_ready$\{T_A\}$, $\langle$name$\rangle$_retryCount$\{\langle$retryCount$\rangle \cdot T_A\}$

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$\rangle$_P2T	$[\tau_e + \langle$timeoutSeconds$\rangle, \tau_e + \langle$timeoutSeconds$\rangle]$
$\langle$name$\rangle$_P2F	$[\tau_e + \langle$timeoutSeconds$\rangle + \varepsilon, \tau_e + \langle$timeoutSeconds$\rangle + \varepsilon]$
$\langle$name$\rangle$_T2P	$[\tau_e + \langle$retryDelaySeconds$\rangle, \tau_e + \langle$retryDelaySeconds$\rangle]$
$\langle$name$\rangle$_F2R/_C2R	$[\tau_e, \tau_e]$

Fig. 1: Translation pattern of a `RETRY` timeout-policy worker with `FIXED` retry-logic. *Non-urgent* transitions are depicted in gray.

its *Time Reachability Graph* (*TRG*) [8]. The *TRG* construction is automated by the GRAPHGEN software tool [8, 9, 14]. It basically relies on a *symbolic state* notion: each reachable state is a pair: $S = (M, C)$, where M (symbolic marking) maps places into elements of $Bag(TS)$ (*i.e.,* multisets of timestamps) and C (constraint) is a logical predicate formed by linear inequalities defined in terms of $TS \cup \{T_L, T_A\}$, where the symbol T_L represents the state creation instant, and the symbol T_A represents an anonymous timestamp (*i.e.,* a timestamp whose time value does not influence the evolution of the system). The constraint C contains *relative* time dependencies between timestamps. An example of initial symbolic state for the model in Figure 1b can be $S_0 = (M_0, C_0)$, such that:

$$M_0 := \langle \texttt{name} \rangle \texttt{\_schedule}\{T_0\}, \langle \texttt{name} \rangle \texttt{\_retryCount}\{3 \cdot T_A\}$$
$$C_0 := T_0 \geq 0 \ \wedge \ T_0 \leq 10 \ \wedge \ T_L = T_0.$$

Since T_A symbols are unessential for the computation of firing times associated with enabled transitions, they do not appear in the symbolic constraint C_0.

Given the *TRG* structure, model checking algorithms can be applied to verify the correctness of the system against requirements expressed as specific *Time Computation Tree Logic* (TCTL) properties [15, 16]. The model checking technique is fully automated by the GRAPHGEN software tool.

3 A Running Example: the Taxi-hailing Application

To illustrate the use of CONDUCTOR2PN, we consider a taxi-hailing application example, such as Uber [17]. This application has a modular architecture: at the

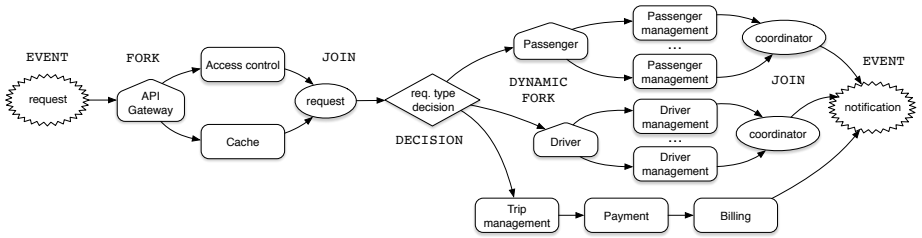


Fig. 2: High level schema of the taxi-hailing orchestration.

core is its business logic, which is implemented by modules that define domain objects and events. Surrounding the core are adapters (*e.g.,* database access components, messaging components, *etc.*) that interact with the external world, and web components that either expose APIs or implement a user interface (UI). Many organizations in this context are re-engineering their monolithic applications to adopt microservice architectures. The idea is to split the application into a set of smaller cohesive, independent process services interacting via messages.

A possible decomposition of the taxi-hailing system is shown in Figure 2. This schema follows the notation introduced in [4] and shows all the services and the overall workflow. It can be automatically generated from the Conductor blueprint (by using the Conductor framework).

Each microservice (*i.e.,* a rectangular shape in Figure 2) implements a (micro) functionality (*e.g.,* access control, trip management, payment, etc.) and is deployed independently, possibly into cloud virtual machines or Docker containers [18]. Moreover it exposes a REST API consumed by other microservices or by the application's clients. For instance, the *Passenger management* uses the *Notification* service to notify a passenger about an available driver. The *API gateway* instead, exposes a public API used by mobile clients or web UIs.

Other shapes represent control and data flow primitives (*e.g,* `EVENT`, `FORK`, `JOIN`, *etc.*) executed within the Conductor orchestration server to manage the execution and scalability of the entire process flow. For instance, *req. type decision* allows to choose between alternative flows depending on the request type. The *Passenger* and the *Driver* components use the dynamic fork primitive to dispatch user requests to different (possibly replicated) services, geographically located in different areas. A complete list of all available workflow primitives of Conductor is available in Table 2. They will be described more in depth in section 4.2.

It is worth noting that this example is intended to represent a fictional but significative microservice system. We have designed it to highlight several features supported by our formal framework.

4 Time Basic Net Semantics for Conductor

This section introduces the semantics we defined for microservice process flows. The semantics is defined by giving the translation from the Conductor blueprint

into a TB net formal model. We provide a translation for each construct of the Conductor language, *i.e.,* a JSON-based domain specific language that describes both the involved microservices (*Worker task*s) and the execution flow (*System task*s). The model generation process is guided by the identification of *translation pattern*s of each individual component. Each pattern has *input*/*output* elements for joining it with other patterns. The final model is the composition (*i.e.*, the union by joining input/output elements) of different TB net patterns of the corresponding microservices and execution flow constructs. The translation process is fully automated by the Conductor2Pn[1] Java software tool.

4.1 Worker tasks

Worker tasks represent the individual microservices. Worker tasks can be implemented in any language and talk to the Conductor via REST API endpoints to poll for other tasks and update their status after execution. Our modeling approach abstracts away from implementation details and it views a single worker as a black-box component. We use TB net places to represent the state of a task and TB net transitions to represent the task primitives. Moreover, we make use of temporal functions associated with transitions to specify temporal constraints upon scheduling and execution.

Figure 1a represents an example of worker task metadata definition using the Conductor domain specific language. This example contains a JSON object which lists a number of fields used to tell the Conductor engine how to manage the microservice lifecycle. Accordingly, we apply the corresponding translation pattern (Figure 1b). In this example, a timeout (*i.e.,* the `timeoutSeconds` property) has been set. Thus, if the `payment` task does not complete within 1200 milliseconds, Conductor retries the task again (*i.e.,* the `RETRY timeoutPolicy`) up to 3 times (*i.e.,* the `retryCount` property). Upon timeout, the task is rescheduled after a fixed delay (*i.e.,* the `FIXED retryLogic`) of 600 milliseconds (*i.e.,* `retryDelaySeconds` property). The tasks need up to 200 milliseconds to be scheduled (*i.e.,* `scheduleSeconds` property).

Figure 1b depicts the translation pattern of the worker task listed in Figure 1a. The pattern can be instantiated by replacing each ⟨`field`⟩ with the corresponding value read from the JSON object. Dashed line shapes represent the *input* elements. Therefore, the `payment` task is being executed whenever the `payment_schedule` place become marked. Double line shapes represent *output* elements. In this example, the `payment_complete` is the element used to join the `payment` task to other subsequent patterns.

Table 1 contains the complete description of metadata associated with worker tasks. Fields marked as *user defined assumption* are not part of the Conductor language, but represent additional information used by our translation process to define the user assumption on time required by specific lifecycle operations.

[1] Conductor2Pn has been released as open-source software. It is available for download at: `https://bitbucket.org/seresearch_unimi/conductor2pn`.

Table 1: Metadata associated with worker tasks.

field	description	notes
name	worker task name.	Unique.
retryCount	#retries to attempt when a task is marked as timed out.	-
timeoutSeconds	Timeout (msec.) to complete a task after transiting to inProgress status.	No timeouts if set to 0.
timeoutPolicy	Task's timeout policy.	Possible values: `RETRY`, `ALERT_ONLY`, `TIME_OUT_WF`.
retryLogic	Mechanism for the retries.	Possible values: `FIXED`, `EXPONENTIAL_BACKOFF`.
scheduleSeconds	Time (msec.) needed to schedule a task.	User defined assumption.
terminateSeconds	Time (msec.) needed to end a task upon termination request.	User defined assumption.

If not defined (*e.g.,* no prior information is available, or they have negligible values), a default value is supplied. Different combinations of values assigned to `timeoutPolicy` and `retryLogic` fields determine different semantics and different corresponding translation patterns, as follows.

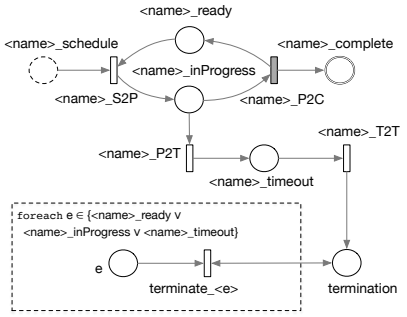
– The `RETRY timeoutPolicy` along with `FIXED retryLogic` is described above and the corresponding translation pattern is shown in Figure 1b.

– The `ALERT_ONLY timeoutPolicy` means that the worker task just registers a counter upon timeout. The translation pattern can be easily obtained starting from Figure 1b and by erasing the places: ⟨`name`⟩`_retryCount`, ⟨`name`⟩`_timeout`, ⟨`name`⟩`_retryUsed`, and the transitions: ⟨`name`⟩`_T2P`, ⟨`name`⟩`_F2R`, and ⟨`name`⟩`_C2R`, (along with incoming/outgoing edges). Meaning that the task does not retry the execution, but it goes into a failure state after the first timeout.

– The `TIME_OUT_WF timeoutPolicy` means that the entire workflow is marked as timed out and terminated upon worker's timeout. This translation pattern is shown in Figure 3a. The dashed box represents a `foreach` macro substitution and it means that the inner elements are repeated for each element `e` (*i.e.,* for each ⟨`name`⟩`_ready`, ⟨`name`⟩`_inProgress` and ⟨`name`⟩`_inProgress` places, in this specific pattern).

– The `RETRY timeoutPolicy` along with `EXPONENTIAL_BACKOFF retryLogic` means that upon timeout, the task is rescheduled after `retryDelaySeconds` multiplied by the attempt number. This translation pattern is shown in Figure 3b. Likewise the `RETRY-FIXED` pattern (Figure 1b), the generated model retries to execute a timed out task a fixed number of times, but with a different timeout foreach retry attempt. The `for` macro substitution is used here to replicate the inner elements so that each retry attempt is constructed with the correct temporal constraint. The place ⟨`name`⟩`_retryUsed` along with transitions ⟨`name`⟩`_C2R` and ⟨`name`⟩`_F2R` are used to implement the same mechanism used in `RETRY-FIXED` tasks to refill the ⟨`name`⟩`_retryCount` place, once a termination state is reached.

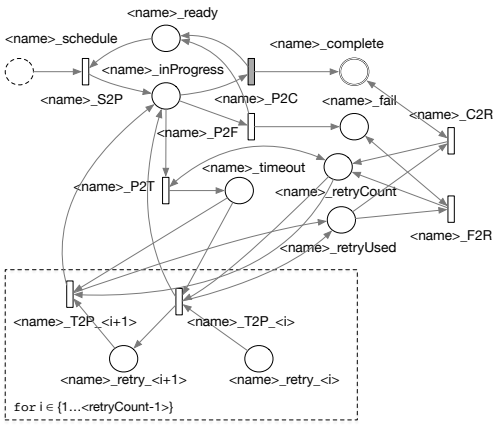
The TB net models of the individual microservices' lifecycle represent the basic components that are joined together with the system task models.



Initial marking: ⟨name⟩_ready$\{T_A\}$

Transition	**Time function**
⟨name⟩_T2T	$[\tau_e, \tau_e + \langle\text{terminateSeconds}\rangle]$
tarminate_⟨e⟩	$[\tau_e, \tau_e]$
other	See Figure 1.

(a) `TIME_OUT_WF` pattern.



Initial marking: ⟨name⟩_ready$\{T_A\}$, ⟨name⟩_retryCount$\{\langle\text{retryCount}\rangle \cdot T_A\}$, ⟨name⟩_retry_1$\{T_A\}$

Transition	**Time function**
⟨name⟩_T2P_⟨i⟩	$[\tau_e + \langle\text{retryDelaySeconds}\rangle \cdot \langle i\rangle,$ $\tau_e + \langle\text{retryDelaySeconds}\rangle \cdot \langle i\rangle]$
other	See Figure 1.

(b) `EXPONENTIAL_BACKOFF` pattern.

Fig. 3: Translations of a `TIME_OUT_WF`, and a `RETRY EXPONENTIAL_BACKOFF` worker tasks, respectively. *Non-urgent* transitions are depicted in gray.

4.2 System tasks

The overall process flow in Conductor is a sequence of worker tasks (denoted, in the Conductor language, by the `SIMPLE` value associated with the `type` field) and system tasks (Table 2 lists all possible system task types). System tasks represent the execution flow primitives and their execution/scalability is managed by the Conductor engine. Our translation process defines a formal semantics for all system tasks, however, for the sake of space, we describe in the following the translation pattern of some representative examples used in our taxi-hailing application.

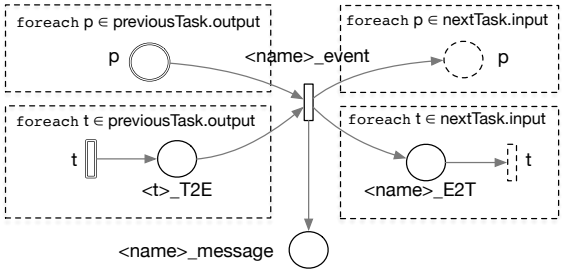
Event Task — This system task publishes an event (*i.e.,* a message) to either Conductor or an external system. Messages to Conductor can create event based dependencies for workflows and tasks by using event handlers. Handlers execute specific actions (*i.e.,* either start a workflow, fail a task, or complete a task) when a matching event occurs. In our taxi-hailing application example we make use of an event handler to start the workflow upon a *request* event. Moreover, an event task (*i.e., notification*) is used whenever a core functionality must notify a user. Figure 4a lists an example of an event task defined by using the Conductor language. It requires the following configuration parameters: the `inputParameters` is a map where keys are parameters' reference name and

Table 2: Description of all available system tasks.

type name	purpose
DYNAMIC	A worker task which is dynamically derived based on the input to the task, rather than being statically defined.
DECISION	Similar to the `switch case` statement in a programming language.
FORK	Fork is used to schedule a parallel set of tasks.
FORK_JOIN_DYNAMIC	Same as fork, except that the list of tasks to be forked is provided at runtime using task's input. A JOIN task must follow the dynamic fork.
JOIN	Join task is used to wait for completion of multiple tasks spawned by a (dynamic) fork tasks.
SUB_WORKFLOW	Sub Workflow task allows for nesting a workflow within another workflow.
WAIT	A wait task is implemented as a gate that remains in inProgress state unless marked as completed or failed by an external trigger.
HTTP	An HTTP task is used to make calls to another microservice over HTTP.
EVENT	Publish an event (message) to either Conductor or an external system. They are useful for creating event based dependencies for workflows and tasks.

```
{ "name": "notification",
  "inputParameters": {
   "dst": "${input.dst}",
   "msg": "${input.msg}"
  },
  "type": "EVENT",
  "sink": "sqs:messageQueue"
  "eventSeconds": "800"
}
```

(a) Event task JSON object.



(b) Translation pattern.

Initial marking: $\emptyset$

Transition	Time function
⟨key⟩_decision	$[\tau_e, \tau_e + \langle \text{eventSeconds} \rangle]$

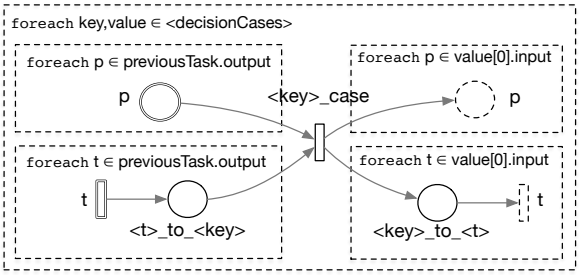
Fig. 4: Translation pattern of an `EVENT` system task.

values are inputs parameters of the event tasks; the `sink` represent the recipient of the event which can be either Conductor or an external system like Amazon Simple Queue Service (SQS) [19] used in our example; the `eventSeconds` defines an additional assumption on the time required by the event generation process (*i.e.,* 800 milliseconds in our example). Figure 4b shows the corresponding translation pattern. It connects all output elements of the previous task (*i.e.,* `previousTask.output`) to the ⟨`name`⟩`_event` transition that produces the event (*i.e.,* a new token into the ⟨`name`⟩`_message`). The postset of this transition also includes of all the input elements of the subsequent task (*i.e.,* `nextTask.input`). Tokens in the ⟨`name`⟩`_message` place can be consumed by the different event handlers, if any.

Decision task — A decision task represents a `switch-case` like statement. In our taxi-hailing application, it is used to decide over alternative process flows, depending on the user type of an incoming request. An example is shown in Figure 5a. The `caseValueParam` is the name of the parameter in task input

```
{ "name": "functionalArea",
  "type": "DECISION",
  "inputParameters": {
    "value": "${req.Type}"
  },
  "caseValueParam": "value",
  "decisionCases": {
    "Driver": [ ... ],
    "Passenger": [ ... ],
    "Trip": [ ... ]
  },
  "decisionSeconds": "600"
}
```

(a) Decision task JSON object.



(b) Translation pattern.

Initial marking: $\emptyset$

Transition	**Time function**
⟨key⟩_decision	$[\tau_e, \tau_e + \langle \text{decisionSeconds} \rangle]$

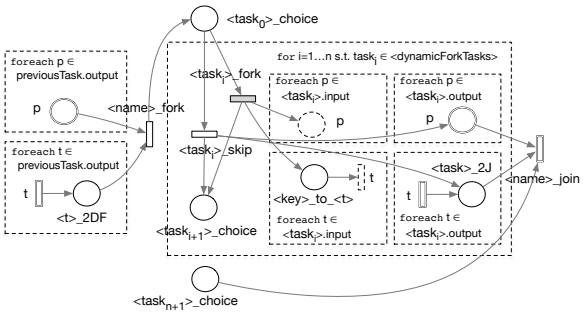
Fig. 5: Translation pattern of a `DECISION` system task.

whose value will be used as a switch; the `decisionCases` is a map where keys are possible values for `caseValueParam` and values are lists of tasks to be executed; the `decisionSeconds` defines an additional assumption on time required by the decision process. Figure 5b shows the translation pattern of a decision task. Since the `decisionCases` in Figure 5a has three `key-value` pairs, by applying the external macro substitution we obtain three transitions: `Driver_case`, `Passenger_case`, and `Trip_case` which represent a non-deterministic choice between three different alternative execution flows. The output elements of the previous task in the process flow (*i.e.,* `previousTask.output`) represent the preset of these transitions, while the input elements of the first task in the value list (*i.e.,* `value[0].input`) represent their postset.

Dynamic Fork Join task — Dynamic fork join tasks are used in our taxi-hailing application to dispatch requests to the appropriate services, depending on the geographical location of the user. The request may be dispatched to multiple replicated services to increase the resilience. Figure 6a shows an example which defines two system tasks: the *dynamic fork* and the *join* (that must always follow the former one). The `dynamicForkTasksParam` is the name of the parameter in task input whose value contains the list of tasks to be executed in parallel. The `inputParamName` is a map where keys are forked task's reference name and values are inputs parameters of forked tasks. The `dynamicForkTasks` represents an additional user defined assumption used to identify at design time the set of tasks that can be launched during the execution of the dynamic fork. The example in Figure 6a shows that `Passenger` task can spawn any combination of worker tasks chosen from the following multiset: $\{3 \cdot \texttt{PassengerManagement}\}$, *i.e.,* the taxi-hailing application can dispatch the request to up to three `PassengerManagement` tasks to deal with the user requests. If the `dynamicForkTasks` is not defined, the default assumption is $\{1 \cdot \texttt{task}\}$ for each `task` in the worker tasks set. The `dynamicForkSeconds` and

```
{ "name": "Passenger",
  "type": "FORK_JOIN_DYNAMIC",
  "dynamicForkTasksParam":
  "dynamicTasks",
  "inputParamName":
  "input",
  "dynamicForkTasks": [
    "3*PassengerManagement"
   ]
  "dynamicForkSeconds": 250
},
{ "name": "coordinator",
  "type": "JOIN"
  "joinSeconds": 250
}
```

(a) A dynamic fork followed by a join task JSON objects.



(b) Translation pattern.

Initial marking: $\emptyset$

Transition	**Time function**
⟨name⟩_fork	$[\tau_e, \tau_e + \langle \text{dynamicForkSeconds} \rangle]$
⟨name⟩_join	$[\tau_e, \tau_e + \langle \text{joinSeconds} \rangle]$
⟨task$_i$⟩_fork/_skip	$[\tau_e, \tau_e]$

Fig. 6: Translation pattern of a `FORK_JOIN_DYNAMIC` system task. *Non-urgent* transitions are depicted in gray.

the `joinSeconds` define additional assumption on time required by the dynamic fork and join processes, respectively.

Figure 6b shows the translation pattern of a dynamic fork task followed by a join task. The initial component generated by the translation process is the `Passenger_fork` transition. The preset of this transition is composed of all the output components of the previous task. The external `for` macro substitution (on the right side), allows the ⟨`task`$_\mathtt{i}$⟩`_fork`/`_join` components to be replicated for each `task`$_\mathtt{i}$ in the `dynamicForkTasks` multiset. Thus, whenever the place `task`$_\mathtt{i}$`_choice` is marked, the `task`$_\mathtt{i}$ can either be executed or not, depending on the non deterministic choice of fire either ⟨`task`$_\mathtt{i}$⟩`_fork` or ⟨`task`$_\mathtt{i}$⟩`_skip`. Whenever all the forked tasks complete their execution, the `Passenger_join` transition becomes enable to fire (*i.e.,* the output component of the translation pattern).

The final model, automatically derived from the translation process of Conductor2Pn, formally defines the entire execution flow and can be used to perform different verification activities, such as simulation and model checking.

5 Formal Verification

Based on the TB net modeling formalism described in the previous sections, we are able to formally verify the requirements by essentially inspecting the *TRG* structure. In this Section, we describe some verifiable properties, by means of

some significative examples upon the taxi-hailing application. The properties can be verified by using the CONDUCTOR2PN output directly fed into GRAPHGEN.

A very common property to check is *deadlock/livelock* freedom. If the property does not hold, all the paths leading to a deadlock state can be easily visualized. Livelock freedom has been proven on the taxi-hailing process flow. However, there exist potentially unwanted execution paths leading into deadlock states. For instance, since the *access control* microservice is not replicated, it represents a single point of failure. When, for some reason, it would not be reachable, the incoming requests would be partially handled and the process flow would not reach a final state. This scenario is represented by a feasible execution path, where the *non-urgent* transition `accessControl_P2C` never fires.

More generally, it is possible to formalize the requirements by using specific CTL/TCTL formulas [20, 21], in order to verify *invariant*, *safety*, *liveness* and *bounded-response time* properties [8, 16]. An example of a *safety* property is:

$$\neg EF(\texttt{payment\_inProgress} > 0 \wedge \texttt{payment\_timeout} > 0) \tag{1}$$

Formula (1) means that does not ($\neg$ operator) exists (E operator) a feasible path, where the argument eventually (F operator) holds. The argument is a state formula expressed as combination of conditions on the number of tokens in places: *i.e.,* an inconsistent state of the `payment` component, where both `inProgress` and `timeout` status coexist, a condition that we want to be sure it will never happen. An example of a *liveness* property is:

$$\begin{aligned} AG(&\texttt{accessControl\_complete} > 0 \wedge \texttt{cache\_complete} > 0 \implies \\ &AF(\texttt{Passenger\_schedule} > 0 \vee \texttt{Driver\_schedule} > 0 \vee \texttt{TripManagement\_schedule} > 0)) \end{aligned} \tag{2}$$

Formula (2) is used to verify that for all paths (A), globally (G), if a request has been handled, then a decision between `Driver`, `Passenger` and `Trip` management tasks is always (A), eventually (F) taken.

Bounded-response time properties can be used to perform timing analysis on the process flow. A simple example of this property is presented in Formula 3:

$$AG(\texttt{payment\_schedule} > 0 \implies EF_{\leq 2400}(\texttt{billing\_complete} > 0)) \tag{3}$$

This formula is used to verify that whenever a payment task is requested, it is possible to complete the billing process (without failures), within 2.4 seconds. Another example follows below:

$$\begin{aligned} AG(&\texttt{APIGateway\_forking} > 0 \wedge \texttt{notification\_message} = 0 \implies \\ &EF_{\leq 4800}(\texttt{notification\_message} > 0)) \end{aligned} \tag{4}$$

Formula 4 is used to verify that whenever a user request is received, a final state (*i.e.,* a notification message has been enqueued), is reachable in 4.8 seconds.

6 Related Work

Orchestration [3] and choreography [22] are two well-established alternative approaches to define cooperation between services in order to provide arbitrary complex interactions and functionalities. Although, orchestration was more popular in SOA, this approach has recently seen a renewed interest due to its

simplicity of use and easier ways to manage complexity. Moreover, peer task choreography, can lead to harder scalability with growing business needs [4]. The approach presented in this paper has been mainly influenced by different related lines of work aiming at reuse theoretical results from well-established models and techniques in formal methods. Recent proposed techniques try to connect choreographies and behavioral types to either logic-based formalisms, such as μ-calculus [23], linear logic [24], or automata-based formalisms, such as abstract machines and communicating automata [25]. Our translation approach is somehow inspired by previous studies that propose to map Business Process Execution Language (BPEL) for Web Services [26–28] to PNs. These techniques provide computer aided verification for SOA systems. However, these approaches are not directly applicable in the context of microservices, where new emerging languages and frameworks, such as Jolie [29] and Conductor [4], are being adopted as major references for orchestration, from both industry and academia.

Our work represents the first attempt (to the best of our knowledge) to leverage the expressiveness of the TB nets to supply formal specification and verification of microservice based process flows defined via Conductor blueprints. TB nets also allows to easily map and analyze temporal aspects that can be of primary importance in different application contexts, at least to ensure a certain quality of service. The formalization process opens up the possibility to directly apply model checking, simulation, model-based testing, and runtime verification by means of powerful off-the-shelf tools, such as Graphgen [8, 14], MaRDiGraS [30] and MahaRAJA [31].

7 Conclusion and Future Work

In this paper we propose a formal semantics for process flows specified using Conductor, *i.e.,* an open source orchestration framework in use at Netflix, Inc. Our approach aims at mechanically producing a formal representation in terms of Time Basic Petri Nets. The translation process is fully automated by means of a Java software tool called Conductor2Pn. The formal semantics is complete (*i.e.,* it covers all the language constructs of Conductor). The TB net model can be used to perform model checking, simulation, model-based testing, and runtime verification by means of powerful off-the-shelf tools. We demonstrated the use of the tool on a small taxi hailing system, illustrating how to analyze the behavior of a Conductor blueprint by model-checking its translation.

We are interested in expanding on this work in different directions. Conductor2Pn is currently a prototypal implementation and has been tested on a variety of small benchmarking examples. We are going to test it with more sophisticated real-world applications in order to evaluate the scalability of the proposed approach. Moreover, we want to expand the translation to stochastic formalisms to support both performance analysis and probabilistic model checking in presence of uncertainty. A suitable modeling formalism that nicely supports these features is Generalized Stochastic Petri Nets [32].

References

1. J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," https://martinfowler.com/articles/microservices.html, mar 2014.
2. 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. [Online]. Available: https://arxiv.org/pdf/1606.04036.pdf
3. M. Mazzara and S. Govoni, "A case study of web services orchestration," in *Proceedings of the 7th International Conference on Coordination Models and Languages*, ser. COORDINATION'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 1–16. [Online]. Available: http://dx.doi.org/10.1007/11417019_1
4. "Netflix, Inc. Conductor," https://netflix.github.io/conductor/, accessed: June 2017.
5. The Netflix Tech Blog, "Netflix conductor: A microservices orchestrator," https://medium.com/netflix-techblog/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40, dec 2016.
6. C. Ghezzi, S. Morasca, and M. Pezzè, "Validating timing requirements for time basic net specifications," *J. Syst. Softw.*, vol. 27, pp. 97–117, November 1994.
7. J. L. Peterson, "Petri nets," *ACM Comput. Surv.*, vol. 9, no. 3, pp. 223–252, Sep. 1977. [Online]. Available: http://doi.acm.org/10.1145/356698.356702
8. C. Bellettini and L. Capra, "Reachability analysis of time basic Petri nets: A time coverage approach," in *Proceedings of the 2011 13th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, ser. SYNASC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 110–117.
9. M. Camilli, "Petri nets state space analysis in the cloud," in *Proceedings of the 2012 Int. Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1638–1640.
10. J. Bengtsson and W. Yi, *Timed Automata: Semantics, Algorithms and Tools.* Springer Berlin Heidelberg, 2004, pp. 87–124.
11. Y. Gurevich, "Sequential abstract-state machines capture sequential algorithms," *ACM Trans. Comput. Logic*, vol. 1, no. 1, pp. 77–111, Jul. 2000.
12. 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.
13. D. G. D. L. Iglesia and D. Weyns, "Mape-k formal templates to rigorously design behaviors for self-adaptive systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 10, no. 3, pp. 15:1–15:31, Sep. 2015.
14. M. Camilli, C. Bellettini, L. Capra, and M. Monga, "Coverability analysis of time basic Petri nets with non-urgent behavior," in *18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2016, Timisoara, Romania, September 24-27, 2016*, J. H. Davenport, V. Negru, T. Ida, T. Jebelean, D. Petcu, S. M. Watt, and D. Zaharie, Eds. IEEE Computer Society, 2016, pp. 165–172. [Online]. Available: https://doi.org/10.1109/SYNASC.2016.036
15. R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for real-time systems," in *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, Jun 1990, pp. 414–425.
16. M. Camilli, A. Gargantini, and P. Scandurra, "Specifying and verifying real-time self-adaptive systems," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th Int. Symposium on*, Nov 2015, pp. 303–313.

17. "Uber Technologies, Inc." https://www.uber.com/, accessed: June 2017.
18. D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241
19. "Amazon Simple Queue Service," https://aws.amazon.com/sqs/, accessed: June 2017.
20. R. Alur, C. Courcoubetis, and D. Dill, "Model-checking in dense real-time," *Inf. Comput.*, vol. 104, no. 1, pp. 2–34, May 1993.
21. 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.
22. C. Peltz, "Web services orchestration and choreography," *Computer*, vol. 36, no. 10, pp. 46–52, Oct 2003.
23. L. Caires and F. Pfenning, "Session types as intuitionistic linear propositions," in *Proceedings of the 21st International Conference on Concurrency Theory*, ser. CONCUR'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 222–236. [Online]. Available: http://dl.acm.org/citation.cfm?id=1887654.1887670
24. P. Wadler, "Propositions as sessions," in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '12. New York, NY, USA: ACM, 2012, pp. 273–286. [Online]. Available: http://doi.acm.org/10.1145/2364527.2364568
25. L. Caires and J. A. Pérez, "Multiparty session types within a canonical binary theory, and beyond," in *36th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems - Volume 9688*. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 74–95. [Online]. Available: https://doi.org/10.1007/978-3-319-39570-8_6
26. S. Hinz, K. Schmidt, and C. Stahl, *Transforming BPEL to Petri Nets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 220–235. [Online]. Available: http://dx.doi.org/10.1007/11538394_15
27. N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg, "Analyzing interacting ws-bpel processes using flexible model generation," *Data Knowl. Eng.*, vol. 64, no. 1, pp. 38–54, Jan. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.datak.2007.06.006
28. C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede, "Formal semantics and analysis of control flow in ws-bpel," *Sci. Comput. Program.*, vol. 67, no. 2-3, pp. 162–198, Jul. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2007.03.002
29. 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. [Online]. Available: https://doi.org/10.1016/j.entcs.2007.01.051
30. C. Bellettini, M. Camilli, L. Capra, and M. Monga, "Mardigras: Simplified building of reachability graphs on large clusters," in *Reachability Problems*, ser. LNCS, P. Abdulla and I. Potapov, Eds. Springer, 2013, vol. 8169, pp. 83–95.
31. 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 International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, ser. LNCS, C. Barrett, M. Davies, and T. Kahsai, Eds. Cham: Springer International Publishing, 2017, vol. 10227, pp. 115–130.
32. M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, *Modelling with Generalized Stochastic Petri Nets*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1994.