Equivalence-Enhanced Microservice Workflow Orchestration to Efficiently Increase Reliability

Zheng “Jason” Song and Eli Tilevich
Software Innovations Lab, Virginia Tech
{songz,tilevich}@cs.vt.edu

Abstract—**The applicability of the microservice architecture has extended beyond traditional web services, making steady inroads into the domains of IoT and edge computing. Due to dissimilar contexts in different execution environments and inherent mobility, edge and IoT applications suffer from low execution reliability. Replication, traditionally used to increase service reliability and scalability, is inapplicable in these resource-scarce environments. Alternately, programmers can orchestrate the parallel or sequential execution of equivalent microservices—microservices that provide the same functionality by different means. Unfortunately, the resulting orchestrations rely on parallelization, synchronization, and failure handing, all tedious and error-prone to implement. Although automated orchestration shifts the burden of generating workflows from the programmer to the compiler, existing programming models lack both syntactic and semantic support for equivalence. In this paper, we enhance compiler-generated execution orchestration with equivalence to efficiently increase reliability. We introduce a dataflow-based domain-specific language, whose dataflow specifications include the implicit declarations of equivalent microservices and their execution patterns. To automatically generate reliable workflows and execute them efficiently, we introduce new equivalence workflow constructs. Our evaluation results indicate that our solution can effectively and efficiently increase the reliability of microservice-based applications.**

I. Introduction

Service-oriented software development has embraced the microservices architecture [4], dividing a complex software system into coherent and lightweight microservices, each of which performing a cohesive business function. Although traditionally the microservice architecture is used mainly for composing web services/applications[17], emerging application domains, including IoT and edge computing, have started to increasingly apply this architecture as well[23], [27].

If different microservices fulfill the same application requirement, these microservices provide *equivalent* functionalities that can be used in place of each other. Known application patterns that use equivalence include improving reliability via fail-over and reducing latency via speculative parallelism. In the realm of web applications, service equivalence has been applied to select services: choose the one with the optimal QoS features from its equivalent set [26]. Little prior research has focused on simultaneously executing multiple services to improve reliability, as web services are already quire reliable and the additional costs of simultaneous executions cannot be justified by the expected reliability improvements [11].

Unlike web-based microservices, the ones executed in IoT and edge environments often suffer from partial failures and performance bottlenecks, as is expected for distributed execution environments with naturally dynamic and volatile resources. This work adapts the microservice architecture for such unreliable execution environments by systemically supporting the execution equivalence in microservice-based distributed applications.

The support for equivalence in existing microservice-based programming models [19], [13], [15] is limited: they either cannot explicitly express equivalent microservices or cannot efficiently execute them (i.e., minimize the resources consumed by executing workflows containing equivalent microservices). Without intuitive programming support for equivalence, a non-trivial development effort is required to cost-efficiently increase the reliability of a microservice-based application.

In this paper, we describe a dataflow-based programming model that adds support for equivalence in orchestrations of microservice-based applications. Our programming model extends the dataflow programming pattern in [13]: programmers declaratively specify microservices and their dataflow relationships; the compiler automatically generates a workflow that schedules the execution plan for these microservices, with different execution strategies expressed as workflow constructs; and the runtime steers the execution of microservices based on the workflow and their execution results. In particular, we extend the dataflow specifications and workflow constructs with support for equivalent microservices, and provide rules to generate and execute such workflows. Our evaluation demonstrates that our solution simplifies the expression of equivalent functionalities and suites particularly well for adapting distributed executions to dynamic contexts.

As a summary, the contribution of this paper is three-fold:

- We introduce a dataflow-based microservice orchestration language that explicitly supports execution equivalence.
- We introduce workflow constructs for executing equivalent microservices that provide a fine-grained control over the life cycle of microservice execution.
- Through case studies, we demonstrate how our solution can be applied to develop real applications, and how the resulting workflows can increase their reliability cost-efficiently. We also show that our solution outperforms prior approaches in striking the right trade-offs between reliability improvements and resource consumption.

The rest of this paper is organized as follows: Section II introduces the work’s background. Section III analyzes the problems in programming microservice-based applications with equivalence. Section IV gives the design details of our solution, which is evaluated by Section V. Section VI discusses our design choices and Section VII concludes this paper.

II. Background and Related Work

This paper focuses on providing programming support for orchestrating services containing equivalent microservices. In this section, we first discuss how microservice-based applications have taken advantage of equivalence, and then summarize major programming models for engineering such applications.

A. Equivalent Microservices

Hosted at different cloud servers with dissimilar QoS characteristics, various microservices can provide the same functionality. In the research domain of cloud-based microservice composition, such equivalent microservices are referred to as competing microservices [26]. This domain focuses on how to choose a set of services that maximize the overall QoS while satisfying the QoS requirements of each service [3]. Hiratsuka et al. [11] further explore the combined use of functional-equivalent microservices to enhance the QoS. They leverage two general orchestration patterns for equivalent microservices: fail-over for reliability enhancement and speculative parallel for efficiency enhancement. In the edge and IoT domains, Osmotic computing [27] switches between cloud/edge-based microservice deployments to optimize the overall QoS.

In edge/IoT environments, equivalent microservices can also deliver the same functionality. However, these microservices can differ not only in their respective QoS characteristics, but also in the way they are implemented, including the hardware/-software resource utilization, algorithms, and compositions. For example, [16] demonstrates that the environmental temperature can be captured by a temperature sensor, or be inferred from the CPU temperature; both wireless methods [24] and optical methods [18] have been used to obtain the indoor location of individuals. As an edge application is expected to run in dissimilar edge environments that feature different available sensor and computational resources and runtime contexts, it is hard to guarantee the overall reliability given the low reliability of individual microservice executions [25]. The combined use of equivalent microservices can improve reliability while striking a good balance between the response time and costs. Therefore, when extending the microservice architecture to the domains of IoT and edge computing, microservice equivalence can increase the power and expressiveness of existing programming models.

B. Programming Models for Orchestrating Microservices

Workflow languages (e.g., WS-BPEL [19]) are widely used in engineering service-oriented systems and applications, due to their ease of use and ability to manage complexity [8]. A service/application workflow can be represented as a set of microservices and assist workflow control nodes, together with their order of invocation and data passing relationships [2]. In general, the assist workflow nodes consist of a start node, an end node, and any number of repeatable pre-defined workflow constructs. Such constructs represent different workflow control patterns (some researchers use different terms to represent the same concept, e.g., structured activities [19] or operational semantics [5]). At runtime, an execution engine runs workflows by following the operational semantics of the contained workflow constructs.

BPEL is a block-structured workflow description language that helps developers to express and execute workflows. The workflow patterns that can be expressed by BPEL are: sequential processing, conditional behavior (if), repetitive execution (while), selective event processing (pick), parallel processing and processing multiple branches (foreach) [19]. Accordingly, the supported workflow constructs are: sequence, parallel (AND-fork/join), exclusive choice (XOR-fork/join), loop, and multi-choice [21]. Although over 40 workflow patterns (including structured discriminator, which supports speculative parallel execution) have been developed, most of them are not explicitly supported in general workflow orchestration languages, such as BPEL [21].

The sheer number of BPEL features complicates the language's functional semantics. Besides, designed for machine processing, BPEL requires its developers to master graphical composition tools. Several novel domain-specific languages improve programmability. Orc [15] is a structured language, in which service orchestrations are specified by means of functional programming idioms. It also provides semantic support for handling concurrency, time-outs, exceptions, and priority. The workflow patterns in Orc (specified as combinators) support fail-over (`otherwise`) and speculative parallel (`pruning`) execution strategies. Although these strategies can be used to orchestrate the execution of equivalent operations, the language provides no facilities to control the fine-grained lifecycles of these operations.

Both BPEL and Orc require the programmers to specify the control logic of concurrency and failure handling. To further shift the burden of workflow orchestration from the programmer to the compiler, dataflow-based domain-specific languages are introduced for orchestrating workflows automatically [13]. Such DSLs enable the programmers to specify the data dependencies between microservices, and generate the workflow accordingly, for "data dependencies equivalent to scheduling" [14]. Dataflow programming has been widely adopted by IoT/edge computing [9], [22], [20], [10], [6] and stream processing [12]. However, these dataflow languages have no support for equivalence.

III. Problem Analysis

We start analyzing the problem domain by giving two example use cases that demonstrate how equivalent microservices can be used to satisfy reliability requirements. These use cases are a fire detection system and an offline digital store.

A. Use Cases of Leveraging Equivalent Microservices

(1) Use Case 1, `fireDetection`: One important security task of smart homes is detecting fire. Flame sensors have been commonly used in private homes and office buildings

for a long time [1]. However, in those cases in which a flame sensor is temporally unavailable or absent altogether, sensor data fusion can also accurately detect fire. For example, one alternative method can combine a temperature sensor and a camera (two most widely deployed sensors in smart homes) to detect fires [7]. Specifically, if both the smoke density level extracted from captured environment images and the room temperature captured by the sensor exhibit unusually high levels, these two conditions happening simultaneously indicate the presence of fire. Hence, we have two equivalent strategies for detecting fires: (1) read a flame sensor, and (2) (a) read a temperature sensor; (b) capture and process image. As fire detection must be both time efficient and reliable, the strategies (1) and (2) can be executed speculatively parallel to fulfill these requirements.

The use case above can be expressed modularly as individual microservices. Microservice `thresholdCheck` takes as input `firePossibility`, and returns a `boolean` value `isFireDetected`. Microservice `readFlameSensor` takes no input, checks the states of flame sensors, and outputs `confidence`, which can be used as `firePossibility`. Microservice `sensorFusion` takes as input `smokeDensity` and `temperature`, and outputs `firePossibility`. Microservice `getTemperature` queries the temperature sensor and outputs `temperature`. Microservice `getImage` captures images by camera, and outputs `imageUrl`. Microservice `inferSmokeDensity` takes as input `imageUrl`, processes the image and outputs `smokeDensity`. Fig. 1 lists the input/output relationships of these microservices.

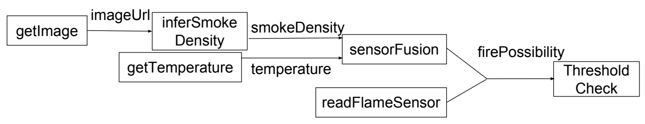

Fig. 1. Data Dependencies of Use Case 1

(2) Use Case 2, `purchaseItemDetection`: In offline digital stores, a customer purchases merchandise by picking up items from shelves; upon exiting the store, the customer's credit card is charged for the purchases. An enabling technology for such stores is *purchase detection*, using sensors to track purchases in real time.

TABLE I
MICROSERVICES USED IN `PURCHASEITEMDETECTION`

Microservice	Input	Output
getBarcodeFromVideo	video	barcode
getItemIDFromBarcode	barcode	itemID
getShelfFromVideo	video	shelfID
estimateItemLocation	video, shelfID	row, line
getItemIDFromLocation	row, line, shelfID	itemID
getWeightChange	shelfID	weight
estimateItemIDbyWeight	shelfID, weight	itemID

The `purchaseItemDetection` application takes a video clip of a purchase as input and outputs the `itemID` of the purchased item. To recognize the purchased item, the application first analyzes the video clip for the item's barcode. If this method fails, it infers the `itemID` by processing sensor data in two equivalent ways: (1) use the shelf's id and the purchased item's location on the shelf to infer what the purchased item is; (2) obtain the delta of weight from the shelf's scale, and based on the delta infer what the purchased item is.

B. Deficiencies of Existing Programming Models

As mentioned in Section I, existing programming models lack explicit equivalence facilities and cannot support the above application scenarios.

(1) No semantic and syntactic facilities for equivalence in dataflow specification languages. An approach presented in [11] provides the pipeline, data distribution and data aggregation patterns. When translated to workflows, pipelines are converted to execute sequentially, and data distributions/aggregations are converted to execute in parallel (AND-fork/join). Hence, this dataflow specification has no semantic and syntactic support for the fail-over and speculative parallel workflow patterns that leverage execution equivalence.

(2) Inability to handle exceptional execution conditions systematically: Consider removing the equivalent microservice `readFlameSensor` to generate use case 1' without any equivalent microservices. Fig. 2 shows how use case 1' can be expressed in a dataflow language. Notice that the dataflow contains no processing rules that specify how to handle execution failures. That is, when a microservice fails, no alternate microservice can be invoked to continue the execution, thus causing the overall execution to terminate.

```
//...binding microservices
getImage -> inferSmokeDensity
inferSmokeDensity -> smokeDensity
getTemperature -> temperature
(smokeDensity,temperature) -> sensorFusion
sensorFusion -> thresholdCheck
```

Fig. 2. Dataflow-based Workflow Specification for Use Case 1'

However, with equivalence, the terminate-by-default failure handling rule no longer applies. For example, if the workflow of use case 1' is extended with an equivalent microservice `readFlameSensor`, the failure of `getTemperature` no longer terminates the overall execution.

(3) Poor cost efficiency: Although prior approaches, including Orc [15] and the" 1 out-of-m join" [21] can express the execution strategies of equivalent microservices, these approaches were not designed to provide a fine-grained control over the life cycle of microservices. When executing workflows with equivalence, this lack of control may lead to using computational resource unproductively. Consider the following examples: 1) `readFlameSensor` finishes its execution at time t_1, while `getImage` and `getTemperature` remain in operation. To steer the execution cost efficiently would necessitate passing `firePossibility` returned by `readFlameSensor` to `thresholdCheck`, and immediately terminating both `getImage` and `getTemperature`. Although the "1-out-of-m join" pattern does execute `thresholdCheck`, it does nothing to stop the now unnecessary execution

of `getImage` and `getTemperature`; 2) `getTemperature` times-out at time t_2, while `getImage` remains in operation. As `sensorFusion` requires both `temperature` and `smokeDensity`, missing either one of these required inputs would make it impossible to execute `sensorFusion`. Hence, efficiency would necessitate terminating `getImage` and waiting for `readFlameSensor` to complete. However, Orc would wait for `getImage` to finish its execution and then proceed to executing `inferSmokeDensity`.

IV. Workflow and DSL for Equivalence

We first introduce special equivalence-supporting workflow constructs and discuss how they are supported in the runtime. As a specific example of supporting equivalence programmatically, we discuss the design and implementation of our dataflow DSL.

A. Workflow Overview

The main distinction of workflows with equivalence is that the successful/failed execution of microservices may affect the execution status of other microservices currently in operation or to be invoked. In traditional workflow graphs, nodes denote microservices, and a directed edge between them denotes their execution order. Although handling equivalence requires storing and executing additional control flows for each microservice, we maintain the basic structure of traditional workflows, adding to it new workflow constructs and notification rules. Unlike the operational semantics of general workflows, our new workflow control constructs collect the necessary execution states of the microservices within their scope and react accordingly.

B. Workflow Constructs

A workflow graph, representing a service, contains a set of nodes as a set of directed edges, $G =< N, E >$.

Edges: A directed edge $e(n, m, d)$ connects node n to node m, $\forall n, m \in N$, and specifies the data d passed from n to m. We call n predecessor and m successor.

Nodes: $N = N_{ms} \cup N_{control}$, where N_{ms} denotes a set of microservice nodes and $N_{control}$ denotes a set of control nodes. $N_{control}$ contains one service start node $start$, one service end node end, and any number of pairs of workflow control nodes. For any microservice node in N_{ms}, $n(i, o, timeout)$ maintains its required input i, generated output o, and allowed execution time $timeout$.

Workflow Patterns: $C(k)$ denotes a pair of control nodes, with C_{start}, C_{end}, and k representing the start node, the end node and the start node's out-degree, respectively. A pair of control nodes can be one of the following three types:

- parallel, where k represents the number of concurrent branches. A parallel pair starts all k branches simultaneously, until all executions complete. If one branch fails, the pair fails, terminating the executions of the remaining branches.
- fail-over, where k represents the number of equivalent branches. It starts one branch at a time and outputs the results of the first successful execution.
- speculative parallel, where k represents the number of equivalent branches. It starts all the k branches simultaneously, and outputs the first obtained result. If one branch succeeds, the pair terminates the executions of the remaining branches.

C. Multi-threading and Execution State

The workflow's execution can be described as a finite state automata of states and transitions [28]. We introduce the operational semantics from the perspective of states and transitions. In particular, we introduce how the states and transitions are combined with concurrency (thereafter, we use threads to demonstrate all general concepts).

A service's execution starts from the root thread. Multiple child threads can be spawned by and joined to one parent thread. Each spawned child thread maintains a parent handle, synchronized across all its siblings, used to notify the parent of whether the child's execution succeeded or failed. Each thread maintains a **`counter`** and a **`currentNode`**. The **`currentNode`** indicates the current node being executed by this thread. An executing thread can be terminated by its **`currentNode`** or interrupted by its parent thread, with the **`currentNode`** handling the interruption. The **`counter`** indicates the number of successfully executed branches if the thread is executing a parallel control pair, the current executed branch for a fail-over pair, and the number of failed branches for a speculative parallel pair.

Our workflow design possesses the following features:

- Each microservice node has only one direct predecessor and one direct successor. In other words, the control node pairs control all the spawning and joining concurrency actions.
- For a pair of control nodes, the start node's out-degree equals the end node's in-degree. Although the control nodes can be nested (e.g., a pair of control node is contained in another pair of control nodes), the number of spawned threads at the start node of the pair equals to the number of joined threads at the end node.

A node's execution status can be in one of the three states: **`running`**, **`succeeded`**, and **`failed`**. The $start$ and end nodes can be only in the **`running`** state. The start nodes of control pairs can be in the **`running`** and **`succeeded`** states, while other nodes can be in any of the three states. A **`running`** state can transition to either **`succeeded`** or **`failed`**. In the **`succeeded`** or **`failed`** state, the **`currentNode`** runs as dictated by the operational semantics of its type, which comprises terminating the current thread, sending notifications to the parent thread, interrupting child threads, and setting the **`currentNode`** to another node.

D. Workflow Operational Semantics

The semantics is implemented by following these state transition rules. If the current state is:

1) running: A microservice node starts the microservice's execution. The node transitions to the **`failed`** state if the execution fails or times out, and transitions to the **`succeeded`** state if the execution succeeds. The $start$ node sets the **`currentNode`** to its successor, and end outputs the results.

The start node of a parallel pair and a speculative parallel pair set the **counter** to k (the number of branches), spawn k child threads, set the **currentNode** of the child threads to the first nodes of these threads, and transition to the **succeeded** state (see Fig. 3.a and Fig. 5.a). The start of a fail-over pair sets **counter** to k if it equals 0, executes the $counter$th branch, and transitions to **succeeded** (see Fig. 4).

The end node of a control pair waits to be signaled by its child threads. Upon receiving a **succeeded** signal, 1) the end of a parallel pair decrements the **counter**, and transitions to the **succeeded** state if **counter** == 0, or maintains the **running** state if **counter**>0 (see Fig. 3.b); 2) the end of a fail-over pair or a speculative parallel pair transitions to the **succeeded** state; Upon receiving a **failed** signal, 1) the end of a parallel pair transitions to the **failed** state; 2) the end of a fail-over or speculative parallel pair decrements the **counter**, and transitions to the **failed** state if **counter**==0 (see Fig. 5.b). If otherwise **counter**>0, the end of a fail-over pair sets the **currentNode** to its pair start (see Fig. 4), and the end of a speculative parallel pair maintains the **running** state. If the thread is interrupted, the end node of its control pair interrupts all child threads in turn.

2) succeeded: 1) A microservice node sets the **currentNode** to its direct successor. If the direct successor is the end node of a control pair, the thread sends a **succeeded** signal to its parent and terminates itself. Otherwise, it continues to execute the new **currentNode**. 2) the start node of a control pair sets the **currentNode** to its pair end node. 3) a pair end node sets **counter** to 0, interrupts all child threads if the pair is speculative parallel, and follows the microservice node's processing rules.

3) failed: 1) A microservice node checks if it is executed by the root thread: if so, it sets the **currentNode** to the end node; otherwise, the thread sends a **failed** signal to its parent and terminates itself. 2) an end node of a control pair sets the **counter** to 0, interrupts all child threads if the pair is parallel, and follows the microservice node's processing rules.

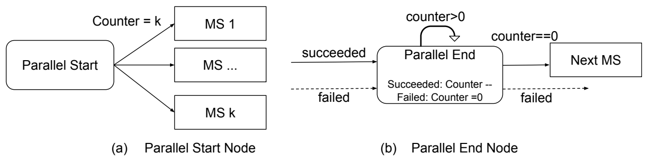

Fig. 3. Parallel Pair State Transition

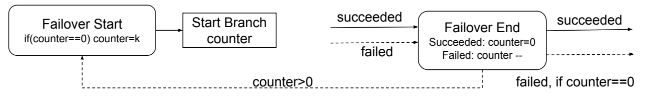

Fig. 4. fail-over Pair State Transition

E. DSL and Workflow Generation

To demonstrate how dataflow specifications can support equivalence, we create a domain specific language, MDLE (**M**icroservice **D**ataflow **L**anguage with **E**quivalence). A

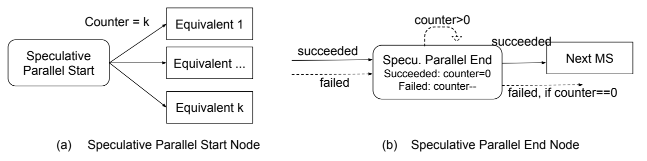

Fig. 5. Speculative Parallel Pair State Transition

MDLE script declaratively specifies a collection of microservices. Aliases implicitly define the data flow between microservices. Multiple microservices providing the same input are considered equivalent.

```
<Service> ::= <ID> <Description>
<ID> ::= "Service "String
<Description> ::= "{"[<Params>] <MSs>"}"
<Params> ::= "input:"|"output:" [<Variable> ","]
<MSs> ::= [Microservice]+

<Microservice>::="MS:"<MSID>"{" [<MSDetail>]+ "}"
<MSID>::=String

<MSDetail>::=<Timeout>|<Input>|<Output>|<Prior>
<Timeout>::="timeout:" [<Select_Rule> "."]+
<Input>::="input": [<MS_input> ","]+
<Output>::="output": [<MS_output> ","]+
<MS_output>::=<output_Variable> ["as" <Alias>]
<MS_input>::=[<Alias> "as"] <input_Variable>
<Alias>::=String
<Prior>:="priority:" "high"|"low"|"medium"
```

Fig. 6. DSL EBNF Definition.

1) MDLE EBNF: Fig. 6 defines the syntax of MDLE in EBNF. Some of the key features are as follows:

- Each service is identified by a unique id, `ID`. `Params` can either be `input`, which must be passed when the service is invoked, or `output`, which is the returned execution result.
- A service comprises `Microservices`, identified by unique `MSIDs`, and containing additional attributes.
- A microservice invocation comprises the following attributes: 1) the `Input` parameters that specify the microservice's invocation parameters; 2) the `Output` that specify what results should be returned, which can be renamed to `Alias`; 3) the `Prior`ity of a microservice, which can be `high`, `medium`, or `low`. Programmers can use the priority parameter to indicate which equivalent microservice should be preferred to provide the required input; 4) the optional `timeout` rules that specify the timeout values for each microservice. If the value is not specified, a default timeout value is used.

```
Service example {
  output: y
  MS: A { output: a as x
     //priority: medium}
  MS: B { output: b as x
     //priority: medium}
  MS: C { output: c as x
     //priority: high}
  MS: D {
     input: x; output: y}}
```

Fig. 7. Example Service Suite

Fig.7 shows an example MDLE script. The original outputs of microservices A, B, C are a, b, c, and are aliased to x. Microservices A, B, and C are equivalent, and unless explicitly specified, their priorities are medium by default. Hence, they should be executed in a speculative parallel way. If line 4, 7 and 10 are uncommented, microservice C and the speculative parallel of A and B should be orchestrated as fail-over.

2) Aliasing Output and Input: A microservice may require multiple inputs, and can generate multiple outputs. To differentiate these inputs and outputs, a microservice developer assigns different names to these inputs and outputs. In a service script, these names are identified as `input_Variable` and `output_Variable`. MDLE uses `alias` to implicitly specify equivalent microservices. Two or more microservices are considered equivalent if their output is set to the same `alias`, which is a required input for another microservice.

3) Compiling a MDLE *Script to a Workflow Graph:* The MDLE compiler converts dataflows into a workflow graph, via a bottom-up procedure. The compiler maintains a dynamic set of nodes to process. The end node is inserted into the set first. Each node in the set is processed in turn, with the newly added nodes replacing the processed ones: 1) if the processed node requires more than one input, the parallel start and end nodes are added to the graph, with each required input becoming a special single-input branch node; 2) if multiple microservices can provide the input of the current node, a pair of corresponding execution control nodes are added, so the data providing microservices become the branch nodes of these control nodes; 3) if only one microservice provides the required input of the current node, it is added to the graph directly. While adding these nodes to the graph, if the current node already has an incoming edge, the new nodes are added between the edge's source node and the current node. The graph generation algorithm terminates once the dynamic set is empty. If the same microservice is invoked in all branches of a control pair, while being directly connected to the pair's start node, the microservice is removed from all branches and added before the pair's start node.

V. Evaluation

We start with a case study of generating and executing a workflow. Then, we assess how our solution improves reliability and efficiency, as compared with workflows without equivalence and those without fine-grained lifecycle control.

A. Case Study

Continuing with a use case from Section III, Fig. 8 shows the MDLE source code of the `fireDetection` service. Our workflow compiler and runtime are implemented in Java. Figs. 9 and 10 show the generated workflows of `fireDetection` and `purchaseItemDetection`, respectively.

Our execution parameters are 80% for the microservice execution success rate and a random number from 1-500ms for the microservice execution time. To demonstrate how the runtime works, we analyze the trace of one execution.

```
Service fireDetection {
   output:isFireDetected
   MS: readFlameSensor {
      output: confidence as firePossibility
      priority: medium
   }
   MS: SensorFusion {
      input: smokeDensity, temperature
      output: firePossibility
      priority: medium
   }
   MS: getTemperature {output: temperature}
   MS: getImage {output: imageUrl}
   MS: inferSmokeDensity {
      input: imageUrl
      output: smokeDensity
   }
   MS: thresholdCheck {
      input: firePossibility
      output: isFireDetected
   }}
```

Fig. 8. Source File of `fireDetection` Service Suite

For use case 1, the 'Speculative Parallel Start' node starts executing at 1ms by forking two threads to execute the 'Parallel Start' node and `readFlameSensor`. At 4ms, the 'Parallel Start' node forks two threads to execute `getTemperature` and `getImage`. At 318ms, `getImage` finishes its execution, with `inferSmokeDensity` continuing on the same thread. At 471ms, `readFlameSensor` finishes its execution, passing a **`succeeded`** signal to the 'Speculative Parallel End' node, waiting on the main thread. Upon receiving the signal, the 'Speculative Parallel End' interrupts its child threads, and then executes `thresholdCheck`. Upon receiving the interrupt, the 'Parallel End' node further interrupts its child threads, thus terminating the execution of `inferSmokeDensity`. At 722ms, `thresholdCheck` finishes its execution, and the 'End' node outputs the result of `thresholdCheck`.

For use case 2, the 'Start' node transitions to the 'fail-over Start' node, which sets the **`counter`** to 2 and the **`currentNode`** to 'fail-over End', spawning a new thread to execute the second branch, `getBarcodeFromVideo`. The microservice fails at 203ms, terminating its thread and sending a **`failed`** signal to 'fail-over End', which transitions to the 'fail-over Start' node to execute the first branch. At 327ms, `getShelfFromVideo` finishes its execution, and the connected 'Speculative Parallel Start' node spawns two threads. At 790ms, the weight change based approach finishes its execution and sends a **`succeeded`** signal to 'Speculative Parallel End', which passes the output to the end node and terminates its child threads, still executing `estimateItemLocation`.

B. Reliability and Cost efficiency

To evaluate the reliability and cost efficiency of our workflow framework, we first set the microservice success rate to 0.8, while varying the average execution time between 100, 200, 300, 400, and 500 (ms). Table II shows the results of 1000 runs for each parameter combination. We observe that the overall successful rate is almost stable. The overall execution

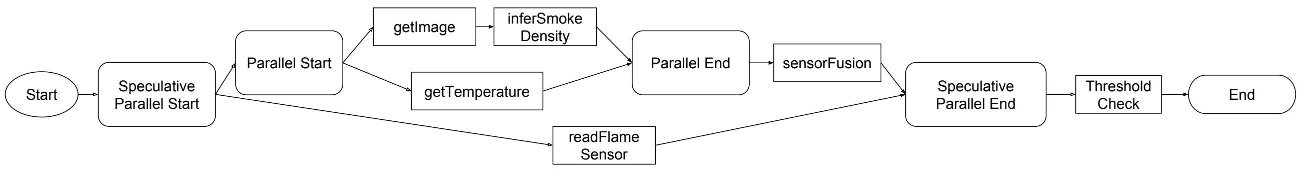

Fig. 9. Generated Workflow for Use Case 1

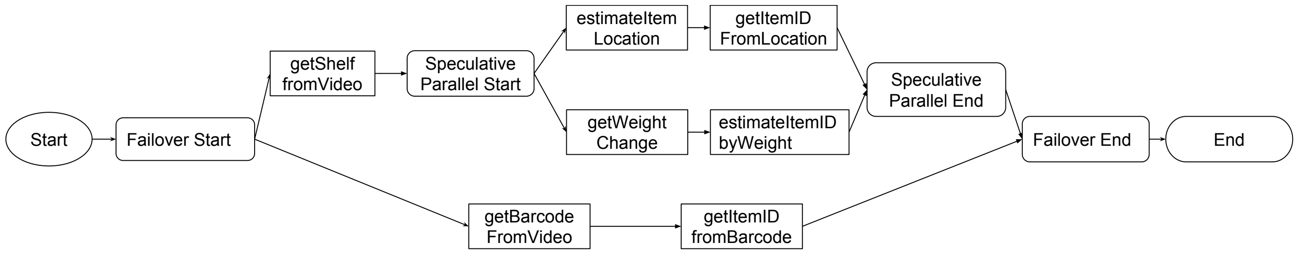

Fig. 10. Generated Workflow for Use Case 2

TABLE II
AVERAGE RESULTS OF 1000 RUNS WITH VARYING EXECUTION TIME

avg execution time	100	200	300	400	500
successful rate	0.714	0.706	0.695	0.721	0.687
ms execution	380	719	1072	1444	1845
finish time	226	433	652	844	1139

TABLE III
AVERAGE RESULTS OF 1000 RUNS WITH VARYING RELIABILITY

reliability	0.2	0.4	0.6	0.7	0.8	0.9
ms execution	542	600	680	710	750	753
finish time	379	398	441	431	443	424
successful rate	0.034	0.193	0.363	0.543	0.718	0.872

TABLE IV
COMPARISON AMONG THREE SOLUTIONS FOR USE CASE 1

	successful rate	execution time	finish time
our method	0.706	380	226
without equivalence	0.484	223	300
without terminating	0.706	523	226

TABLE V
COMPARISON AMONG THREE SOLUTIONS FOR USE CASE 2

	successful rate	execution time	finish time
our method	0.888	308	259
without equivalence	0.555	276	250
without terminating	0.888	320	259

(i.e., cost) and completion times are proportional to the average microservice execution time.

Then, we set the average microservice execution latency to 200ms, varying the microservice success rate between 0.2, 0.4, 0.6, 0.7, 0.8 and 0.9. Table III shows the results of 1000 runs. With the increase of the microservice success rate, the overall reliability and the overall execution time increase. The successful execution of the first microservice causes additional microservice invocations.

We further compare our use case 1 solution with two alternatives: 1) `without equivalence`—randomly execute one of the two equivalent methods; 2) `without terminating`—execute the generated workflow graph without terminating any microservices in operation. We set the microservice reliability to 0.8 and the average latency to 100ms, repeating the simulation 1000 times. Table IV shows that in comparison to `without equivalence`, our solution improves the reliability by 45.9%, reduces the completion time by 24.7%, with the cost of the overall microservice execution time (which can be taken as the resource cost) increasing by 37.7%. Compared to `without terminating`, our solution reduces execution time (i.e., cost) by 27.3%.

We compare use case 2's execution results with those of the aforementioned two alternatives, parameterized identically. Table V shows that compared with `without equivalence`, our solution improves the reliability by 60%, while the overall cost and completion time increase by 11.4% and 3.8%, respectively. Compared to `without terminating`, our solution reduces the execution time (i.e., cost) by 3.9%.

The results of both use cases show that our solution enhances the reliability of microservice-based applications. Due to its fine-grained lifecycle execution control, our solution eliminates the costs of executing microservices that have become unnecessary, a particularly effective optimization for the parallel and speculative parallel execution patterns.

VI. DISCUSSION

Based on the evaluation results, we revisit some of the design decisions behind our workflow and MDLE.

A. *Supported Workflow Constructs*

Although our workflow lacks the "if-else" switch and the "while" loop control patterns, in line with other dataflow-based DSLs [13], we discuss how they can be added to the workflow and MDLE. Adding the "while" loop to the workflow can be treated as a special variant of sequential execution, without spawning any threads. The the switch branches of "if-else" can join in one microservice. For example, to identify a person in a video: if a face image is detected, invoke a face recognition microservice; otherwise, invoke a gait recognition microservice. With the two branches in a "if-else" switch generating the same output, the runtime can execute these switches sequentially. A dataflow-based DSL can encapsulate the "while" conditions within microservices and express the "if-else" switch by conditionally aliasing microservice outputs.

B. Syntactic Support for Equivalence

To support equivalence, programmers must specify: 1) which microservices are equivalent; and 2) how to orchestrate their execution. A dataflow-based DSL can use other alternatives as well. For example, $a * b - c$ can denote that a, b, c are equivalent and orchestrated to execute a, b first speculatively parallel and then execute c if both a, b fail.

However, we choose aliasing and priority to implicitly denote equivalence and orchestrations as: 1) an alias has a unique meaning throughout an application. Aliasing the output in a microservice clearly expresses that the output has its correct physical meaning, thus avoiding programming errors; 2) in the presence of multiple equivalent microservices, programmers only have to decide which microservice's QoS features express the requirements, without having to explicitly orchestrate microservice execution. Our design shifts the burden of orchestrating equivalent microservices from the programmer to the compiler.

VII. Conclusion

We add programming support for equivalence in microservice-based applications by introducing a dataflow-based DSL that extends the notion of dataflow with declarations of equivalent microservices and their execution patterns. Our new equivalence workflow constructs enable the automatic generation of reliable and efficient microservice execution workflows. Supporting equivalence enhances the reliability of microservice-based applications, while our workflow design enhances their cost efficiency.

Acknowledgement

This research is supported by the National Science Foundation through the Grant CCF-1717065.

References

[1] Special issue "sensors for fire detection". https://www.mdpi.com/journal/sensors/special_issues/SFD, 2016. [Online; accessed 13-Feb-2019].

[2] N. R. Adam, V. Atluri, and W.-K. Huang. Modeling and analysis of workflows using petri nets. *Journal of Intelligent Information Systems*, 10(2):131–158, 1998.

[3] M. Alrifai and T. Risse. Combining global optimization with local selection for efficient qos-aware service composition. In *Proceedings of the 18th international conference on World wide web*, pages 881–890. ACM, 2009.

[4] N. Alshuqayran, N. Ali, and R. Evans. A systematic mapping study in microservice architecture. In *Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on*, pages 44–51. IEEE, 2016.

[5] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual modeling of workflows. In *International Conference on Conceptual Modeling*, pages 341–354. Springer, 1995.

[6] B. Cheng, G. Solmaz, F. Cirillo, E. Kovacs, K. Terasawa, and A. Kitazawa. Fogflow: Easy programming of iot services over cloud and edges for smart cities. *IEEE IoT Journal*, 5(2):696–707, 2018.

[7] J. Cheon, J. Lee, I. Lee, Y. Chae, Y. Yoo, and G. Han. A single-chip cmos smoke and temperature sensor for an intelligent fire detector. *IEEE Sensors Journal*, 9(8):914–921, 2009.

[8] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[9] N. K. Giang, M. Blackstock, R. Lea, and V. C. Leung. Developing iot applications in the fog: a distributed dataflow approach. In *Internet of Things (IOT), 2015 5th International Conference on the*, pages 155–162. IEEE, 2015.

[10] N. K. Giang, R. Lea, M. Blackstock, and V. C. Leung. Fog at the edge: Experiences building an edge computing platform. In *IEEE EDGE'18*, pages 9–16. IEEE, 2018.

[11] N. Hiratsuka, F. Ishikawa, and S. Honiden. Service selection with combinational use of functionally-equivalent services. In *Web Services (ICWS), IEEE International Conference on*, pages 97–104. IEEE, 2011.

[12] M. Hirzel and G. Baudart. Stream processing languages and abstractions. *Encyclopedia of Big Data Technologies*, 2018.

[13] W. Jaradat, A. Dearle, and A. Barker. A dataflow language for decentralised orchestration of web service workflows. In *Services (SERVICES), IEEE Ninth World Congress on*, pages 13–20. IEEE, 2013.

[14] W. M. Johnston, J. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM comp. surveys (CSUR)*, 36(1):1–34, 2004.

[15] D. Kitchin, A. Quark, W. Cook, and J. Misra. The orc programming language. In *Formal techniques for Distributed Systems*, pages 1–25. Springer, 2009.

[16] C. Krintz, R. Wolski, N. Golubovic, and F. Bakir. Estimating outdoor temperature from cpu temperature for iot applications in agriculture. In *Proceedings of the 8th International Conference on the Internet of Things*, page 11. ACM, 2018.

[17] A. Leff and J. T. Rayfield. Wso: Developer-oriented transactional orchestration of web-services. In *Web Services (ICWS), 2017 IEEE International Conference on*, pages 714–720. IEEE, 2017.

[18] R. Mautz and S. Tilch. Survey of optical indoor positioning systems. In *Indoor Positioning and Indoor Navigation (IPIN), 2011 International Conference on*, pages 1–7. IEEE, 2011.

[19] OASIS Standard. BPEL 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html#_Toc164738514, 2007. [Accessed 13-Feb-2019].

[20] Y. Qiao, R. Nolani, S. Gill, G. Fang, and B. Lee. Thingnet: A microservice based iot macro-programming platform over edges and cloud. In *2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 1–4. IEEE, 2018.

[21] N. Russell, A. H. Ter Hofstede, W. M. Van Der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. *BPM Center Report BPM-06-22, BPMcenter. org*, pages 06–22, 2006.

[22] L. Safina, M. Mazzara, F. Montesi, and V. Rivera. Data-driven workflows for microservices: Genericity in jolie. In *Advanced Information Networking and Applications (AINA), 2016 IEEE 30th International Conference on*, pages 430–437. IEEE, 2016.

[23] J. M. Schleicher, M. Vogler, C. Inzinger, W. Hummer, and S. Dustdar. Nomads-enabling distributed analytical service environments for the smart city domain. In *2015 IEEE International Conference on Web Services (ICWS)*, pages 679–685. IEEE, 2015.

[24] F. Seco, A. R. Jiménez, C. Prieto, J. Roa, and K. Koutsou. A survey of mathematical methods for indoor localization. In *Intelligent Signal Processing, 2009. WISP 2009. IEEE International Symposium on*, pages 9–14. IEEE, 2009.

[25] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[26] T. H. Tan, M. Chen, J. Sun, Y. Liu, É. André, Y. Xue, and J. S. Dong. Optimizing selection of competing services with probabilistic hierarchical refinement. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 85–95. IEEE, 2016.

[27] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83, 2016.

[28] A. Wombacher, P. Fankhauser, and E. Neuhold. Transforming bpel into annotated deterministic finite state automata for service discovery. In *Web Services, 2004. Proceedings. IEEE International Conference on*, pages 316–323. IEEE, 2004.