From Monolith to Microservices: A Dataflow-Driven Approach

Rui Chen
Software Institute
Nanjing University
Nanjing, Jiangsu, P.R.China
Email: raychennju@gmail.com

Shanshan Li
Software Institute
Nanjing University
Nanjing, Jiangsu, P.R.China
Email: stu.shanshan.li@gmail.com

Zheng Li
Software Institute
Nanjing University
Nanjing, Jiangsu, P.R.China
Email: imlizheng@gmail.com

***Abstract*—Emerging from the agile practitioner communities, the microservice-oriented architecture emphasizes implementing and employing multiple small-scale and independently deployable microservices, rather than encapsulating all function capabilities into one monolithic application. Correspondingly, microservice-oriented decomposition, which has been identified to be an extremely challenging and complex task, plays a crucial and prerequisite role in developing microservice-based software systems. To address this challenge and reduce the complexity, we proposed a top-down analysis approach and developed a dataflow-driven decomposition algorithm. In brief, a three-step process is defined: first, engineers together with users conduct business requirement analysis and construct a purified while detailed dataflow diagram of the business logic; then, our algorithm combines the same operations with the same type of output data into a virtual abstract dataflow; finally, the algorithm extracts individual modules of "operation and its output data" from the virtual abstract dataflow to represent the identified microservice candidates. We have employed two use cases to demonstrate our microservice identification mechanism, as well as making comparisons with an existing microservice identification tool. The comparison and evaluation show that, our dataflow-driven identification mechanism is able to deliver more rational, objective, understandable and consistent microservice candidates, through a more rigorous and practical implementation procedure.**

***Keywords*—microservices, monolith, decomposition, data flow, business logic**

I. Introduction

Monolithic Architecture (MA) [1] is the traditional way for software development, which used to be employed by world-famous Internet services such as Netflix, Amazon and eBay, and it advocates encapsulating all functions in one single application. Less complicated monolithic applications have their own strengths, for instance, simple to develop, test, deploy and scale. Nevertheless, successful applications are always growing in size and will eventually become a monstrous monolith after a few years. Once this happens, disadvantages of the monolithic architecture will outweigh its advantages, for example, overwhelmingly complex and incomprehensible code base of the monolith may hold back bug fixes and feature additions; the sheer size of the monolith can slow down the development and become an obstacle to continuous deployment because of the longer start-up time.

Microservices Architecture (MSA), emerging from agile developer communities, is starting to take shape and now becoming an alternative way that overcomes the challenges of monolithic architecture, by decomposing the monolith into a set of small services and making them communicate with each other through light weight mechanism (i.e. RESTful API). As defined by Martin Fowler et al. [2], microservices architecture is a new architectural style with nine characteristics, e.g., "Componentization via Services". Nevertheless, Newman [3] advocates thinking of microservices as a particular approach for SOA (Service-Oriented Architecture) and also defines microservices by seven principles, including "Model around business concepts", "Decentralize all the things", "Make services independently deployable" [4].

Despite no consensus on the relationship of microservices and SOA currently, the advantages of microservices are commonly accepted both in academia and industry, i.e. maintainability, reusability, scalability, availability and automated deployment [5], [6]. Netflix, Amazon and eBay all have migrated their applications from monolithic architecture to microservices architecture, so as to enjoy the advantages that microservices architecture brings about. For example, Netflix, which is a famous video streaming service, can deal with one billion calls every day now by its streaming API in microservices architecture [7].

The microservices architecture is not a panacea and there are many issues need to be addressed, among which the most important one is how to effectively decompose a monolithic application into a suite of small microservices, for the reason that the decomposition process is usually implemented by hand [8]. Furthermore, the absence of sound evaluation methodology aggravates the challenges of microservice-oriented decompositions.

To address the abovementioned issues, we proposed a purified dataflow-driven mechanism that can guide rational, objective and easy-to-understand microservice-oriented decomposition. Correspondingly, this work makes a threefold contribution: Firstly, this purified dataflow-driven mechanism essentially provides a systematic methodology for microserivce-oriented decomposition. Secondly, compared to the manual implementations [8], our algorithm-driven semi-automated process significantly reduces the complexity in the decomposition practices. Thirdly, the two cases demonstrated in this paper can act as a tutorial to help get familiar with our microservice-oriented decomposition mechanism.

The rest of this paper is organized as follows. Section II briefly summarizes the related work. Section III introduces the design and implementation of the data flow diagram driven mechanism we proposed for microservices decomposition. In Section IV, we elaborate our case studies on microservice-oriented decomposition by employing the dataflow-driven approach. The evaluation of our decomposition mechanism and some limitation discussions are specified in Section V. In Section VI, we draw conclusions of this study and also list potential future work directions.

II. Related Work

A. Microservice-oriented Decomposition

Microservice-oriented decomposition is a prerequisite in migration to microservices architecture. The decomposition process can be represented by the Y-axis of the scalable cube [9]: split the application into small chunks to achieve higher scalability. As mentioned above, decomposing monolithic applications into microservices can bring many benefits more than scalability in theory, while the commonly manual and complex decomposition process [8] will prevent the practices from achieving those benefits. Consequently, it is of great importance to overcome this challenge and realize more effective microservice-oriented decomposition.

In the past three years, some researchers have devoted to addressing the problems of partitioning the application into microservices. Richardson [10] provided four decomposition strategies, i.e. "decompose by business capability", "Decompose by domain-driven design sub domain", "Decompose by verb or use case" and "Decompose by nouns or resources". To the best of our knowledge, the former two are the most abstract patterns which require human involvement and a decision making [11]; while the latter two are easier to realize automation as long as criteria have been predefined. Moreover, as mentioned by Vresk et al. [12], an application might use a combination of verb-based and noun-based decomposition. Disappointingly, this method was not implemented in their study. Hassan et al. [13] offered an architecture-centric modeling concept for microservitization, without mapping the microservices constituents in models to concrete microservices source code and verifying its feasibility. Gysel et al. [14] proposed an service decomposition method on the base of 16 coupling criteria extracted from literature and industry experience. Clustering algorithms are applied to decompose an undirected, weighted graph, which is transformed from Software System Artifacts (SSAs). Yet some processes of the decomposition approach, especially score the edge of the aforementioned graph, lack objectivity. Besides, the transformation from SSAs to predefined, structured input increases the complexity in microservice-oriented decomposition.

In contrast, the microservice-oriented decomposition approach proposed in our work is driven by data flow diagrams from business logic. Based on the objective operations and data extracted from the real-world business logics, our decomposition approach can deliver more rational, objective, and easy-to-understand results.

B. Beyond Microservice-oriented Decomposition

Service-oriented (de)composition in traditional SOA has been a hot topic for decades [15], [16], with two differences to decomposition under microservices architecture. One is that services are coarse-grained in SOA, but usually more fine-grained in microservices architecture. The other one is that decomposition in SOA often aims at selecting the optimal composed service from all possible service combinations regarding some quality requirements, which is a bottom-up process; while in MSA the decomposition procedure may cover the top-down partition first and then the bottom-up integration.

As a top-down method, Data Flow Diagram (DFD) [17] is a good choice for representing monolithic applications from the perspective of business processes [18]. Adler [19] presented an algebra formalizing the decomposition procedure of data flow diagrams, however, it has some drawbacks like inefficiency and no guarantee on finding a good decomposition. Arndt et al. [20] improved Adler's work by replenishing new criteria and implementing their own algorithm for the decomposition of data flow diagrams. Both of these two studies equally lack assessments on decomposition result except for the intuitive notion.

Compared with the abovementioned studies, we transformed the traditional DFD to a purified and detailed version, rather than other SSAs [14] that are difficult to be transformed objectively for decomposition. Moreover, we designed an algorithm to support the two phases' automation in our decomposition approach: (1) condensing the purified DFD to a decomposable DFD and (2) identifying microservice candidates from the decomposable DFD. We also conducted two case studies of microservice-oriented decomposition, and demonstrated the effectiveness of our decomposition approach by evaluation and comparison to an existing decomposition tool.

III. Dataflow-Driven Microservice-oriented Decomposition Mechanism

As mentioned previously, to address the challenge and reduce the complexity in microservice-oriented decomposition, we developed a semi-automatic mechanism to break business logics into microservice candidates. This mechanism comprises three steps including one manual step of dataflow purification and two automatic steps of dataflow-driven decomposition, as specified below.

Step-1. **Construct a purified DFD.**
As part of the requirement analysis, a purified DFD is constructed manually to illustrate the detailed data flow according to some business logic extracted from users' natural-language descriptions. The construction of a purified DFD can start from establishing a traditional DFD and applying our predefined rules (cf. Section III-A2). In particular, the so-called purified DFD focuses on data's semanteme and operation

only, while excluding the side information like data store and external entities that are usually included in traditional DFDs.

Step-2. **Condense the purified DFD into a decomposable DFD.**
By combining the same operations with the same type of output data, the function *GetDecomposableDFD(PDF)* in Algorithm 1 can automatically condense the detailed dataflow specified in the purified DFD into a decomposable dataflow. It is noteworthy that the combination of the output data from the same operations in the purified DFD essentially realizes the improvement of reusability of the potential microservices delivered in the next step.

Step-3. **Identify microservice candidates from the decomposable DFD.**
Given a previously-generated decomposable DFD, the function *GetMicroservices(DDF)* in Algorithm 1 can automatically extract individual modules of operation and its output data from the decomposable data flow. The separation of the modules eventually completes and represents the dataflow-driven microservice-oriented decomposition, while each module indicates a potential microservice to be developed or considered.

We further elaborate the two phases in the following subsections.

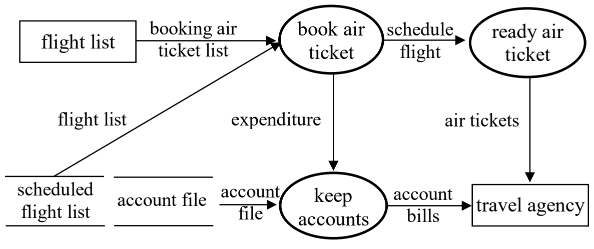


Fig. 1. A example of DFD for booking air ticket system [18].

A. From Traditional DFD to Purified DFD

It is clear that DFD plays a fundamental role in our microservice-oriented decomposition work. By graphically displaying the flow of data within an information system, DFD has widely been used as one of the preliminary tools of requirement analysis in software engineering. Therefore, here we start from introducing a brief background on traditional DFDs, followed by explaining the purified DFD defined in our study.

1) Traditional DFD: From the data's perspective, a DFD, which can be generated from business logic of a system, describes the flow of data through business functions or processes. In other words, it illustrates how data is processed by a business function or an operation of a system in terms of inputs and outputs. Four basic elements comprise a DFD, namely Process, Data Store, Data Flow and External Entity.

Algorithm 1 Microservice-oriented Decomposition Algorithm

```
Input: PDF: the purified DFD
Output: Candidate microservice list
function GetDecomposableDFD(PDF)
                          ▷ combine same operations and its output data
    $DS \leftarrow \emptyset$           ▷ $DS$ : the data set of decomposable DFD
    $PS \leftarrow \emptyset$           ▷ $PS$ : the operation set of decomposable DFD
    for $P \in PDF.operation\_set$ do
        if $P \notin PS$ then
            new $P_n$                          ▷ $P_n$ : new operation
            new $CD_n$               ▷ $CD_n$ : new composed data node
            $CD_n.input \leftarrow CD_n.input \cup P_n$
            $P_n.input \leftarrow P_n.input \cup P.input$
            $P_n.output \leftarrow CD_n$
            $DS \leftarrow DS \cup CD_n$
            $PS \leftarrow PS \cup P_n$
        else
            $P_e \leftarrow PS[i]$                     ▷ $P_e$ : exist operation
            $CD_e \leftarrow P_e.output$    ▷ $CD_e$ : exist composed data node
            $CD_e \leftarrow CD_e \cup P.output$
            $P_e.input \leftarrow P_e.input \cup P.input$
        end if
    end for
    new $DDF$                     ▷ $DDF$ : new decomposable DFD
    $DDF.data\_set \leftarrow DS$
    $DDF.operation\_set \leftarrow PS$
    return $DDF$
end function

function GetMicroservices(DDF)
          ▷ identify microservices from decomposable data flow diagram
    $SS \leftarrow \emptyset$                          ▷ $SS$ : service set
    for $P \in DDF.operation\_set$ do
        new $S$                              ▷ $S$ : new service
        $S.operation \leftarrow P$
        $S.data\_set \leftarrow P.output$
        $SS \leftarrow SS \cup S$
    end for
    return $SS$
end function

return GetMicroservices(GetDecomposableDFD(PDF))
```

Furthermore, traditional DFD mainly has two types of visual representations, which are proposed by Constantine et al. and Gane et al. [21] respectively. However, there is no obvious distinction between these two visualizations, except the representation of processes. In our work, the former type from Constantine et al. is selected as the representation of DFDs.

To facilitate understanding of the traditional DFD, an example of booking air ticket system [18] is shown in Fig. 1. It displays the four basic elements, whose explanations are introduced as follows.

- **Process Notations** (or Operation Notations) refer to activities that operate data of the system. An operation is depicted as a circle with a unique name in form of verb or verb phrase, for example, "book air ticket" and "keep accounts" in Fig. 1. Different from the circle type representation from Constantine et al., operations in DFDs defined by Gane et al. are squares with rounded corners.
- **Data Store Notations** represent the repository of data manipulated by operations, which can be databases or files ("scheduled flight list" and "account file" in Fig. 1). A data store is represented by a rectangle in a DFD with

a name in the form of noun or noun phrase.

- **Data Flow Notations** are directed lines indicating the data from or to an operation. The relevant data sit on the line of a data flow. At least one point of a data flow is linked to a circle of operation.
- **External Entity Notations** stand for the objects which are the destinations or sources of data outside the system. An example of external entity in Fig. 1 is "travel agency". Similar to the representation of data store notation, an external entity is also depicted as a closed rectangle in DFDs.

2) Purified DFD: Purified DFD is a more data-focused version against traditional DFD, by excluding the side information such as data store and external entity from the traditional dataflow representation. As showed in Fig. 2, data in purified DFD is represented by rectangles and data flows only indicate the direction of data transmission. Moreover, operations in purified DFD should be the most fine-grained and regular, and the regularity check will be introduced in next subsection. Therefore, in the potentially purified DFD, we split up the coarse-grained process "book air ticket" (cf. Fig. 1) which has two operations of "flight scheduling" and "cost calculation".

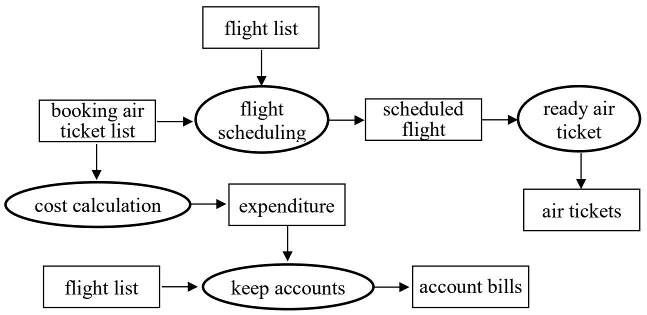


Fig. 2. A example of the potentially purified DFD for booking air ticket system.

As mentioned previously, the construction of a purified DFD can start from drawing a traditional DFD, and then the traditional DFD can gradually be refined by specifying the operation activities and excluding side information (e.g., data store and external entities). In particular, the construction needs to conform to the following two basic rules.

- **Rule 1.** In a purified DFD, operations need to be detailed enough to represent all the individual data processing activities in the original business logic, while the data representation related to an operation should keep the semantic granularities of input and output data without further splits.
- **Rule 2.** In a purified DFD, operations should be normative verbs or verb phrases that can reflect the semantic meaning of the corresponding data processing activities, while data should be named using semantically meaningful nouns or noun phrases occurred in the original business logic.

B. *From Purified DFD to Decomposable DFD*

As described previously, a purified DFD exactly represents the real information flow driven by the corresponding business logic. A decomposable DFD, as a continuation, condenses the precedent purified DFD into a decomposition-friendly dataflow representation by applying a set of rules (e.g., combining the same operations with the same type of output data). Note that, to reduce possible human mistakes when drawing a decomposable DFD and also to make the drawn decomposable DFD more traceable, the backend purified DFD acts as an irreplaceable bridge between the real-world business logic and the corresponding decomposable DFD.

To facilitate explaining the rules of generating abstract DFD, we use a graph definition to conceptualize the purified DFD in advance, as showed in Definition 1. It is also noteworthy that the rules have essentially been implemented in the function *GetDecomposableDFD(PDF)* in Algorithm 1.

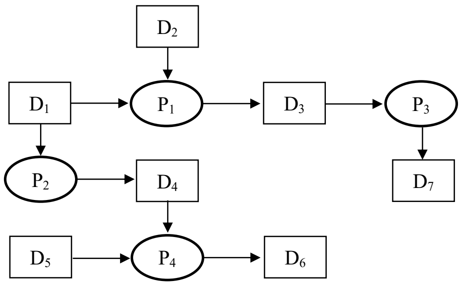


Fig. 3. A conceptual graph of the potentially purified DFD.

Definition 1. A purified DFD is a directed graph $G(V, E)$, where V is the set of nodes and E is the set of directed edges showing the flowing directions of data. In particular, $V = P \cup D$, where P indicates the processing (operation) nodes and D denotes the data nodes. As such, we can use conceptual diagrams to significantly simplify the representation of data flows. For example, given the aforementioned air ticket booking system, by using $P = \{P_1, P_2, P_3, P_4\}$ and $D = \{D_1, D_2, D_3, D_4, D_5, D_6, D_7\}$ to represent the three operations and seven data nodes respectively, the purified DFD (cf. Fig. 2) can be further redrawn into its conceptual version, as illustrated in Fig. 3.

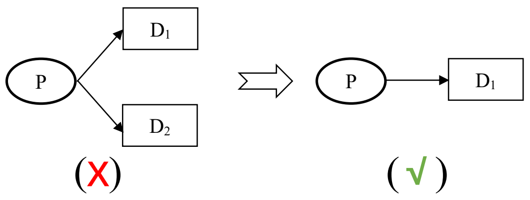
Following Definition 1, we draw conceptual diagram pieces to concisely explain three dataflow condensing rules.

- **Rule 3.** When condensing a purified DFD into a decomposable DFD, each operation node needs to be adjusted to have one output data only.
- **Rule 4.** When condensing a purified DFD into a decomposable DFD, each data node needs to be adjusted to have one type of precedent operation at most.

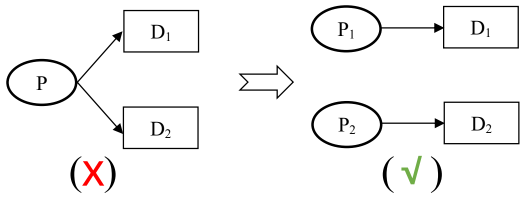
Rule 3 and Rule 4 are applied to ensure the regularity of purified DFDs. In Fig. 4 and Fig. 5, we display scenarios which break these two rules respectively and then provide the regular construction.

According to Rule 3, each operation only has one output data, while data D_1 and D_2 are all output from operation P

in the scenario of Fig. 4. If D_1 and D_2 are the same to each other, then they should be merged into data D (cf. Fig. 4a); otherwise, the operation P should continue to split up into P_1 and P_2 and output D_1 and D_2 respectively (cf. Fig. 4b).



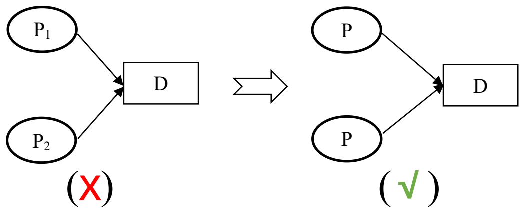
(a)



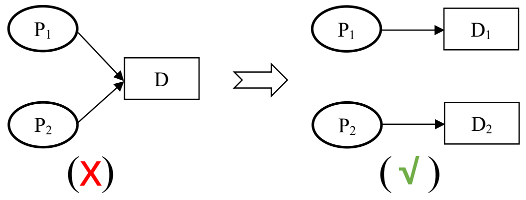
(b)

Fig. 4. An conceptual representation of Rule 3.

On the base of Rule 4, each data needs to be generated by only one type of operation, but two operations P_1 and P_2 output one data D in Fig. 5. In this scenario, P_1 and P_2 should be adjusted to one type of operation P, if they are the same to each other (cf. Fig. 5a); or data D should continue to be broken down into D_1 and D_2, as shown in Fig. 5b.



(a)



(b)

Fig. 5. An conceptual representation of Rule 4.

- **Rule 5.** When condensing a purified DFD into a decomposable DFD, after applying Rule 3 and 4, the same operations with the same type of output data need to be combined into one operation with its output data. Since it is usually difficult to keep the same semantic meaning of the output data after combination, it would be needed to name the combined output data with an abstract semantic concept.

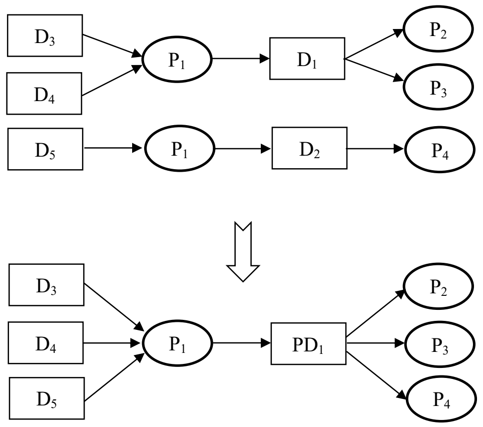


Fig. 6. An conceptual representation of Rule 5.

An example scenario is shown in Fig. 6 to demonstrate how to apply Rule 5 for a decomposable DFD. After combing the same two operations P_1 and its output data D_1 and D_2, the initial input data of P_1—D_3, D_4 and D_5, are all combined into PD_1. Then according to Rule 5, the new output data PD_1 should point to the operations—P_2, P_3 and P_4 into which D_1 and D_2 originally input.

IV. Case Study

To demonstrate and also validate our microservice-oriented decomposition mechanism, we conducted a pair of case studies by employing two typical business logic modules from a recent movie-information-crawling project. When it comes to their data flows, in particular, each operation has only one input data in the first case, while multiple input data might be involved in an operation in the second case.

A. Case Study 1

1) Business Scenario: In the first case, we selected the business logic scenario in which the details and comments of movies are crawled from webpage. To be more specific, as shown in Fig. 7, the application crawls the movie list by an initial url first, then extracts the url of each movie in the list and crawls the webpage, then crawls the comment list for the comment details extraction after extracting the details of a movie and its comment list url, and at last exports structured movie and comment details into database.

The first case study was performed to formally trail the microservice-oriented decomposition process by our three-step mechanism as follows.

2) Constructing a purified DFD: Given the business logic scenario, we manually constructed a purified and detailed DFD according to Rule 1 and Rule 2 (see Section III-A2). Recall that the purified DFD (cf. Fig. 8) excludes the side information such as data store and external entity from the traditional DFD. On the other hand, every single data processing activity of the business logic is displayed in Fig. 8. Based on Rule 1, we kept the original semantic granularity of data as it is, for example, *movie list*. Moreover, we denominated operations with three

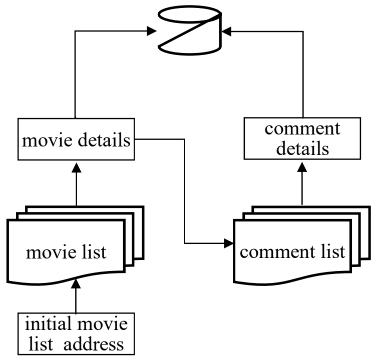


Fig. 7. The business logic scenario of the first case.

types of detailed semantic verbs, i.e. **crawl**, **extract** and **parse**, which represent the corresponding activities in the aforementioned business logic. We also reserved the semanteme of data and represented them with the form of semantic noun phrases, for example, *movie list urls* and *comment details*.

In particular, the purified DFD in Fig. 8 illustrates the details of circularly extracting *movie list urls* and crawling *movie list*. Because movies are usually displayed by multiple pages, the application crawls *movie list* using *movie list urls* of the first page, then extracts the *movie list urls* of next page from the previous *movie list* page, and crawl the next *movie list* pages iteratively. Similarly, it also needs to iteratively extracting *comment list urls* and crawling *comment list* in the business logic of this case.

3) Condensing the purified DFD into a decomposable DFD: After constructing the purified DFD, the regularity should be checked based on Rule 3 and Rule 4 (see Section III-B) to ensure that each operation node has one output data only and each data node has one type of precedent operation at most. It is clear that our purified DFD conforms these two rules, especially when the data *comment list urls* is delivered by two operations under the same type.

To condense the purified DFD into a decomposable DFD, we identified and combined the following three operations with the same type of output data:

- **Operation 1: Crawl.** With the input data *movie list url*, *movie urls* and *comment list url*, the operation Crawl outputs data *movie list*, *movie page* and *comment list* respectively.
  The output data of Crawl are the detailed *webpage content*, which can be obtained by different methods, i.e. URL and Ajax.
- **Operation 2: Extract.** In this case, the operation Extract is used six times. Two Extract instances are to iteratively extract *movie list url* and *comment list url* from the respective input data *movie list* and *comment list*. Two instances use the output data of Crawl (i.e. *movie list* and *movie page*) as the input and then extract the *movie urls* and *comment list url*. The other two instances are performed to extract *movie details* and *comment details* respectively from *movie page* and *comment list* that are the output data of Crawl.
  As for the data, *movie list url*, *comment list url*, *movie urls*, *movie details* and *comment details* are all *partial content* that we want to extract. In other words, they belong to the same type with slight differences in their webpage positions.
- **Operation 3: Parse.** Based on the data *movie details* and *comment details* extracted by the previous operation, Parse occurs twice for the final *structured movie details* and *structured comment details*.
  Both *structured movie details* and *structured comment details* from Parse are then defined as the same type of *structured content*.

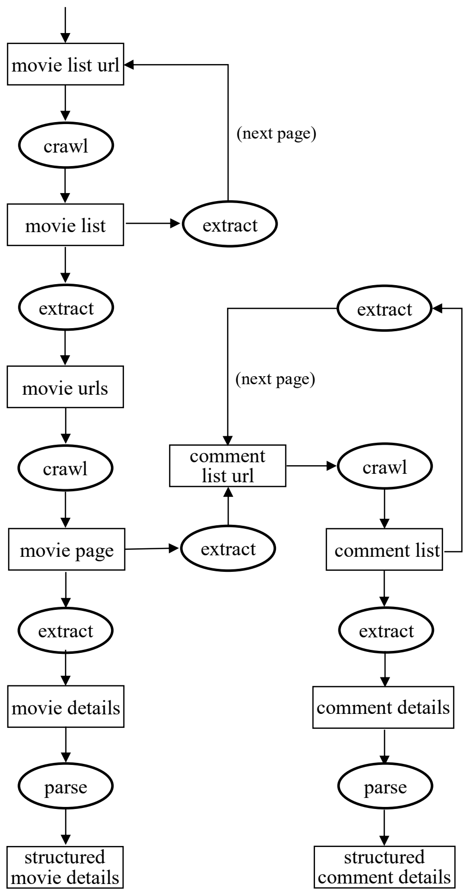


Fig. 8. The purified DFD of the first case.

Therefore, according to Rule 5 (see Section III-B), operations with the same type of output data are combined into one and their output data are denominated by the type with higher-level abstract semanteme (cf. Fig. 9). In particular, the operation combination is implemented automatically by the function *GetDecomposableDFD(PDF)* in Algorithm 1.

4) Identifying microservice candidates from the decomposable DFD: Given a decomposable DFD generated from the purified DFD, we automatically extracted individual operation-

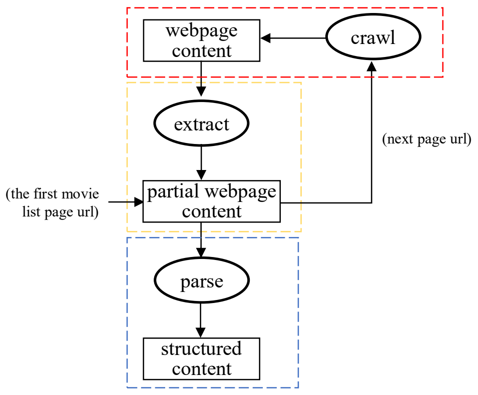


Fig. 9. The decomposable DFD of the first case.

output-data modules from the decomposable dataflow. These individual modules indicate the potential microservices that can be developed independently.

Finally, three microservice candidates were extracted from the decomposable DFD, i.e. "crawl + webpage content", "extract + partial content" and "parse + structured content", which are displayed by dashed boxes in Fig. 9. To facilitate recognition, achieve better independence, and reflect the reusability of services, we finalized the denomination of three microservice candidates as:

- Webpage Crawling
- Text Extraction
- Text Structuring

B. Case Study 2

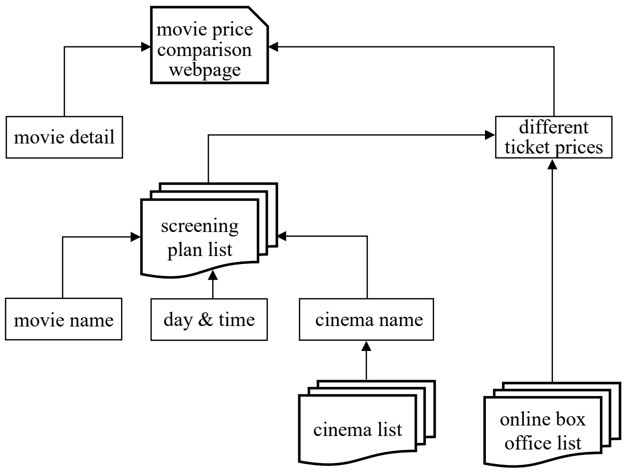


Fig. 10. The business logic scenario of the second case.

1) Business Scenario: In the second case, a business scenario in Fig. 10 originates from a movie ticket price comparison application, which can compare the prices of movie tickets at different online box offices. Specifically, the application extracts each cinema from the cinema list and takes the initial input such as movie name and time from the user to compose one screening information of a movie. Then it queries different ticket prices from a database using the screening information and online box office list. Finally, it integrates the movie details and ticket prices to the final webpage which displays the information of a movie and its ticketing prices from different online box offices.

We conducted the microservice-oriented decomposition process by the following three steps:

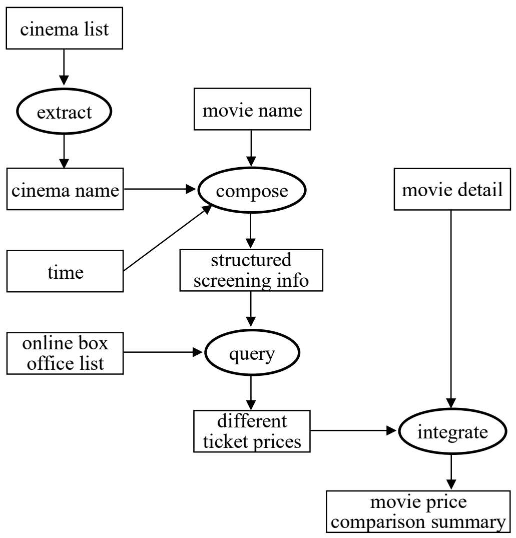


Fig. 11. The purified DFD of the second case.

2) Constructing a purified DFD: We followed the same way to generate the purified and detailed DFD (Fig. 11) as used in the first case. We also removed the external data store and focused on data and operations. We kept the original data as it is according to the Rule 1, for example, *movie name*. Finally, we denominated operations as four types of detailed semantic verbs, i.e. **extract**, **query**, **compose** and **integrate**, from the aforementioned business logic.

Moreover, this case demonstrates a relatively complicated scenario that one operation contains more than one input, e.g., the operation **compose** has three inputs and **query** has two inputs.

3) Condensing the purified DFD into a decomposable DFD: Similar to the first case, we firstly use Rule 3 and Rule 4 to check if our purified DFD conforms predefined rules. For example, although the operation **compose** has three inputs of data, it has only one output data, which makes it compatible with our rules.

Then, we identify the following four types of operations from the purified DFD. Note that there is not any same type of multiple operations in the purified DFD, and thus no operation combination happens at this step.

- **Operation 1: Extract.** This operation here has one input *cinema list* and one output *cinema name*.
- **Operation 2: Query.** This operation uses different input information, i.e. *structured screening information* and

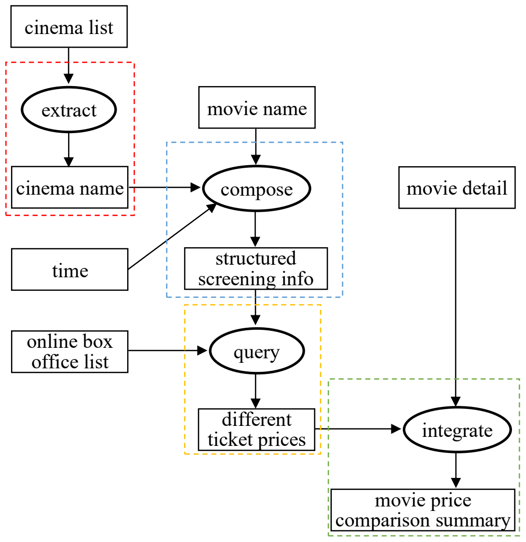


Fig. 12. The decomposable DFD of the second case.

online box office list, to query *different ticket prices* from any given data source.

- **Operation 3: Compose.** This seems to be the most complicated operation which has three input data: *movie name*, *cinema name* and *time*. However, it has only one output data: *structured screening information*, and thus we can keep it as it is.
- **Operation 4: Integrate.** This operation integrates *different ticket prices* and *movie detail* to construct the outcome webpage *movie ticket prices comparison summary*.

In particular, we claim that the operations Compose and Integration cannot be combined together because they take different responsibilities: the former combines single conditions to a structured key used in database queries, while the latter integrates multiple elements into one whole summary webpage without necessarily being structured.

4) Identifying microservice candidates from the decomposable DFD: Finally, following the same way as shown in the first case, we extract four microservice candidates: "extract + cinema name", "compose + structured screening info", "query + different ticket prices" and "integrate + movie price comparison summary" which are displayed by dashed boxes in Fig. 12. To better understand the meaning of these microservices, we renamed them as:

- Text Extraction
- Database Query
- Structure Composition
- Text Integration

It is noteworthy that, among the above four microservice candidates, "Text Extraction" is essentially a reusable one from the first case study.

V. Evaluation and Limitation Discussions

We are concerned with two aspects of microservice-oriented decomposition for evaluating a decomposition mechanism. One aspect is the decomposition result, while another is the decomposition procedure. Therefore, we use a set of microservice principles and a comparison against Service Cutter [14] to validate our decomposition mechanism with regard to those two concerns respectively, followed by a discussion about some limitations of our work at this current stage.

A. Verification against a set of Relevant Microservice Principles

When it comes to microservice principles, both the design-time features (e.g., coupling between microservices) and the runtime features (e.g., scalability against changing workloads) are crucial for evaluating microservice candidates. Since microservice implementations are out of the scope of our decomposition mechanism at this current stage, here we mainly focus on the design-time microservice principles.

1) Fine grain and focus: "Small" and "autonomous" have been considered to be the most important characteristics of microservices [22]. Each microservice is supposed to "do one thing well" [23] to fulfill a focused business responsibility only. By emphasizing specific data processing activities (instead of line of codes or function points), our dataflow-driven mechanism guarantees the most fine-grained microservice candidates in terms of data operation within a business logic. Although small is not necessarily equal to "appropriate" especially when taking into account non-functional trade-offs, the most fine-grained candidates supply a fundamental ground to facilitate further adjusting microservice granularities.

2) High cohesion and loose coupling: As a principle for individual microservices, high cohesion requires every single microservice to implement a relatively independent piece of business logic. In contrast, loose coupling is a crucial principle for involving multiple microservices, which requires the involved microservices to depend barely on each other. Recall that our mechanism extracts "operation and its output data" modules from a decomposable DFD into microservice candidates. By excluding the constraints of input data, each "operation and its output data" module essentially emphasizes its own data processing and does not need to know anything about its precedent and subsequent operations. For example, in the previous Case 1, the intermediate microservice Text Extraction can accept various text sources besides the webpage content, and it cares little about where its output data will go.

3) Neutral development technology: Based on the above-mentioned principles particularly high cohesion and loose coupling, microservices have been claimed to be developable as individual applications with their own delivery pipelines. Although implementing microservices is out of the scope of this study, there is no doubt that our decomposed microservices are able to be realized independently with different programming languages, technologies and even deployment environments. Driven by data flows, our decomposition mechanism enables the candidate microservices to be logically connected through technology-independent data only.

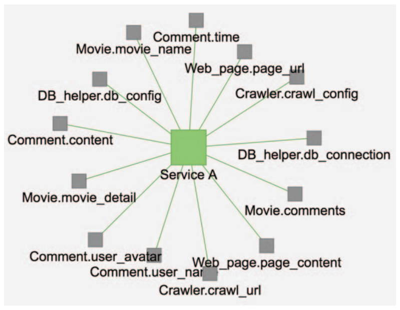


Fig. 13. Decomposition results by Service Cutter for the first case.

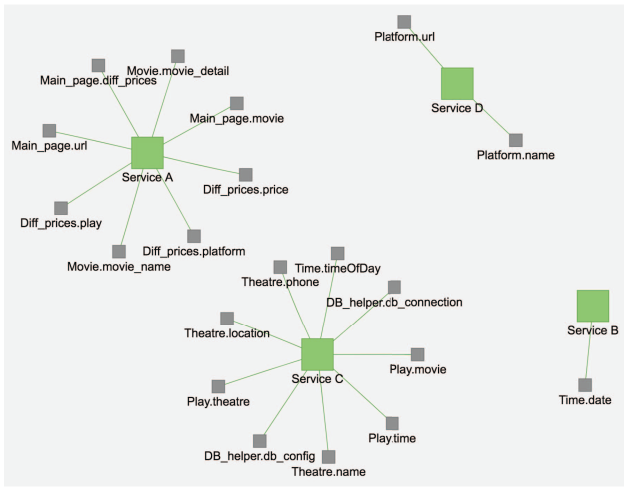


Fig. 14. Decomposition results by Service Cutter for the second case.

B. Comparison to Service Cutter

For further evaluation, we also select a decomposition prototype for microservice identification, i.e. Service Cutter [14], and compare its decomposition results with ours. We select Service Cutter because it is also a semi-automated mechanism and is accessible online. The other related studies (cf. Section II-A) are either manual strategies or closed sources. Therefore, Service Cutter seems to be the most conveniently comparable opponent against our decomposition mechanism.

According to Service Cutter's input requirements and representation criteria, we adjusted the input formats of our case studies, and using Service Cutter to obtain the decomposition results, as shown in Fig. 13 and Fig. 14.[1] In comparison with Service Cutter's decomposition, our approach shows superiorities in the following aspects.

1) More rational: For case 1, Service Cutter only gives one candidate microservice, which would not be fine-grained enough and reusability-friendly. In contrast, the reusability of our operation-focused microservice candidates have inherently been improved since drawing the decomposable DFDs (i.e. combining the same types of data processing activities). In fact, it has been reported that small modularity is one of the keys to reusing a microservice [24]. For case 2, Service Cutter breaks the system into four candidate microservices, which seems reasonable. However, Service B and Service D contain only one or a part of entities. They could have been over cut for being microservice candidates. Moreover, we should not break the entity Time into two microservices, because Time.date and Time.timeOfDay need to stay together to represent a concrete time.

2) More objective: Service Cutter draws many subjective prerequisites into its decomposition. First, Service Cutter relies on the criteria they define to subjectively determine the characteristics of each entity, and then they also use the subjective way to score the weight of dependency graph, which may lead to inaccurate description of the relationship between potential microservices. On the contrary, our approach is more objective, for it is on the basis of DFD which strictly describes the real business logic and its data flow without incorporating many subjective concerns.

3) Easier to operate: Our dataflow-driven mechanism is based on the purified and detailed DFD that is easy to build from the data flow of a real-world business logic. Differently, Service Cutter utilizes Entity Relation Model and many other complex criteria which make the modeling process more complex and not intuitionistic.

4) Easier to understand: Our approach returns the decomposition microservices with the style of "operations + its output data", which is clear and straightforward. More importantly, by largely inheriting relevant semantic meanings from the original data flow, our decomposition mechanism can conveniently generate lightweight descriptions for the resulting microservices [25]. On the contrary, from Service Cutter's result, it is difficult to understand the content of microservices or know how to design them, particularly with data from different entities binding together.

C. Limitations

Despite the aforementioned advantages, our dataflow-driven approach to microservice-oriented decomposition still suffers from several limitations at this current stage.

Firstly, identifying the same data operations requires expertise to some extent. Currently, the combination of the same operations in the purified DFD is based on the operation names. Therefore, it is crucial to denominate operations with suitable semanteme. However, different practitioners might have different naming conventions. To reduce the possible chaos in naming data operations, a potential solution is to define and supply a set of standard operations in advance for drawing purified DFD. Inspired by the basic CRUD operations in database, since we are concerned with the most fine-grained operation activities, there could exist limited common operations in generic data flows.

Secondly, candidate microservices obtained from our decomposition mechanism could still need expert judgement before being developed in practice. As mentioned previously,

[1]The E-R diagrams and JSON-format input data for using Service Cutter in our two case studies are shared online: https://goo.gl/ooo6vo

our mechanism provides the most fine-grained microservice candidates in terms of data operation within a business logic. Unlike Service Cutter, we treat those various predefined criteria to be post-decomposition considerations for adjusting microservice granularities. In other words, given more concerns other than data and operation, some candidates might need to be integrated into relatively coarse-grained microservices at design time. However, by employing our decomposition mechanism, we believe that at least practitioners do not have to worry about further decomposing the generated candidate microservices.

Thirdly, our microservice-oriented decompostion mechanism has not been widely applied to large-scale projects yet. As a result, the comparison between Service Cutter and our mechanism is mainly from the perspective of ourselves. Furthermore, since our proposed mechanism is for design-time decomposition, there is still a lack of evaluation w.r.t. runtime features of candidate microservices (e.g., non-functional requirements like microservices' scalability against changing workloads). To eliminate this limitation, we are currently trying to introduce our decomposition mechanism to several ongoing projects with microservice implementations. As such, we will be able to gradually incorporate practical feedback into our future work to reinforce the evaluation.

VI. Conclusion

In order to address the challenges in migrating monolith to microservices architecture, we proposed a top-down decomposition approach driven by data flows of business logic. The microservice-oriented decomposition process can be break into three phases: First, a purified and detailed DFD is constructed from business logic on the basis of requirement analysis. Second, the purified DFD is automatically condensed into a decomposable DFD in which the same operations with the same type of output data are combined together. Last, microservice candidates are identified and extracted from the decomposable DFD with the description style of "operation + its output data". By using two case studies to evaluate our microservice-oriented decomposition mechanism, we show that both the decomposition results and the decomposition processes are relatively rational, objective, and easy-to-understand.

Considering that our work currently focuses on the design time of microservice-oriented decomposition and correspondingly there are still limitations, we will unfold our future work along three directions. Firstly, we will develop more objective means to constrain the naming convention of data operations. Secondly, we will take into account post-decomposition criteria to help refine the initially generated microservice candidates. Thirdly, we will attend practical projects with microservice implementations, so as to further validate our decomposition mechanism by investigating the runtime performance of the predesigned microservices.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (Grant No.61572251).

References

[1] C. Richardson, "Pattern: Monolithic architecture," 22 June 2017. [Online]. Available: http://microservices.io/patterns/monolithic.html

[2] M. Fowler and J. Lewis, "Microservices," 10 June 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[3] S. Newman, *Building microservices*, 2nd ed. Sebastopol, California: O'Reilly Media, Inc., 27 February 2015.

[4] O. Zimmermann, "Microservices tenets agile approach to service development and deployment," *Comput. Sci. Res. Dev.*, vol. 32, no. 3, pp. 301–310, July 2017.

[5] P. D. Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *Proc. ICSA 2017*. IEEE, April 2017, pp. 21–30.

[6] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *Proc. SOCA 2016*. IEEE, November 2016, pp. 44–51.

[7] C. Richardson, "Introduction to microservices," 9 May 2015. [Online]. Available: https://www.nginx.com/blog/introduction-to-microservices/

[8] G. Kecskemeti, A. C. Marosi, and A. Kertesz, "The ENTICE approach to decompose monolithic services into microservices," in *Proc. HPCS 2016*. IEEE, July 2016, pp. 591–596.

[9] M. L. Abbott and M. T. Fisher, *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*, 1st ed. Boston, Massachusetts: Addison-Wesley Professional, December 2009.

[10] C. Richardson, "Pattern: Microservice architecture," 22 June 2017. [Online]. Available: http://microservices.io/patterns/microservices.html

[11] E. Evans, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[12] T. Vresk and I. Čavrak, "Architecture of an interoperable IoT platform based on microservices," in *Proc. MIPRO 2016*. IEEE, May 2016, pp. 1196–1201.

[13] S. Hassan, N. Ali, and R. Bahsoon, "Microservice ambients: An architectural meta-modelling approach for microservice granularity," in *Proc. ICSA 2017*. IEEE, April 2017, pp. 1–10.

[14] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service Cutter: A systematic approach to service decomposition," in *Proc. ESOCC 2016*. Springer, September 2016, pp. 185–200.

[15] F. Mardukhi, N. NematBakhsh, K. Zamanifar, and A. Barati, "QoS decomposition for service composition using genetic algorithm," *Appl. Soft Comput.*, vol. 13, no. 7, pp. 3409–3421, February 2013.

[16] S. X. Sun and J. Zhao, "A decomposition-based approach for service composition with global QoS guarantees," *Inf. Sci.*, vol. 199, pp. 138–153, March 2012.

[17] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, 1st ed. New Jersey, USA: Prentice-Hall, Inc., 1979.

[18] Y. Zhao, H. Si, and Y. Ni, "A Service-Oriented analysis and design approach based on data flow diagram," in *Proc. CiSE 2009*. IEEE, December 2009, pp. 1–5.

[19] M. Adler, "A decomposition-based approach for service composition with global QoS guarantees," *IEEE Trans. Softw. Eng.*, vol. 14, no. 2, pp. 169–183, February 1988.

[20] T. Arndt and A. Guercio, "Decomposition of data flow diagrams," in *Proc. SEKE 1992*. IEEE, June 1992, pp. 560–566.

[21] C. Gane and T. Sarson, *Structured Systems Analysis: Tools and Techniques*. McDonnell Douglas Systems Integration Company, 1977.

[22] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using Docker technology," in *Proc. SoutheastCon 2016*. IEEE, March 2016, pp. 1–5.

[23] D. Malavalli and S. Sathappan, "Scalable microservice based architecture for enabling DMTF profiles," in *Proc. CNSM 2015*. IEEE, November 2015, pp. 428–432.

[24] T. Schneider, "Achieving cloud scalability with microservices and devops in the connected car domain," in *Proc. Soft. Eng. 2016*. CEUR-WS.org, 23-26 February 2016, pp. 138–141.

[25] J. I. Fernández-Villamor, C. Á. Iglesias, and M. Garijo, "Microservices - lightweight service descriptions for REST architectural style," in *Proc. ICAART 2010*. INSTICC, January 2010, pp. 576–579.