Extracting Service Candidates from Procedural Programs
Based on Process Dependency Analysis

Masahide Nakamura\textsuperscript{1}, Hiroshi Igaki\textsuperscript{1}, Takahiro Kimura\textsuperscript{2} and Ken-ichi Matsumoto\textsuperscript{3}
\textsuperscript{1} Graduate School of Engineering, Kobe University, Japan
\{masa-n, igaki\}@cs.kobe-u.ac.jp
\textsuperscript{2} Nihon Unisys, Ltd., Japan
\textsuperscript{3} Graduate School of Information Science, NAIST, Japan

Abstract

To support legacy migration to service-oriented architecture (SOA), this paper presents a method that derives candidates of SOA services from procedural programs. In SOA, every service is supposed to be a process (procedure) with (1) open interface, (2) self-containedness, (3) coarse granularity for business. We identify such services from the source code and its data flow diagram (DFD), by analyzing data and control dependency among processes. Specifically, we first obtain the DFD with reverse-engineering techniques. For each layer of the DFD we classify every data flow into three categories. Using the data category and control among procedures, we characterize four types of dependency. Finally, we apply six rules that aggregate mutually dependent procedures and extract them as a service. A case study with a liquor shop inventory control system extracts service candidates with various granularity.

1. Introduction

Enterprise software systems are now required to be more agile and flexible to keep up with rapid changes in business environments. However, most enterprise system have been built based on a highly proprietary and monolithic architecture, without considering interoperability among other systems. Such monolithic systems are usually fragile for the changes. A simple update of a business process may result in huge cost for updating the system.

The service-oriented architecture (SOA)\textsuperscript{8}[11] is an architecture paradigm to cope with the problem. In the SOA, features of a system are exhibited as self-contained services, corresponding to elementary business units. A service has open interface that encapsulates implementation-specific logic and data. A business process can be rapidly created and modified by assembling the existing services, where the services are loosely coupled. Thus, the SOA is believed to make the system robust for the business changes.

To receive the benefit of the SOA within the existing assets, the legacy migration to SOA is now a great concern \textsuperscript{6}\textsuperscript{9}. Most of the conventional SOA development frameworks (e.g., \textsuperscript{2}\textsuperscript{3}\textsuperscript{5}) adopt a top-down approach, which starts with the business process analysis, identifies elementary processes, and implements them as services. Since the system is optimimally designed for the SOA, the top-down approach is well applied to development of brand-new systems. However, it doesn’t consider much how to reuse the existing legacy system.

To support the SOA legacy migration effectively, we propose a method that extracts services, as aggregation of processes (procedures), from procedural programs. Specifically, we first obtain the Data Flow Diagram (DFD) \textsuperscript{7} by applying any reverse-engineering techniques to the given source code (e.g., \textsuperscript{1} \textsuperscript{4}). We then classify every data flow in the DFD into three categories (external, system or module). Using the data category, we identify four types of dependency (system data, module data, transaction and condition) between processes. Finally, we aggregate mutually-dependent processes as self-contained services, which is systematically performed by the proposed six rules.

We have conducted a case study with a liquor shop inventory control system. It was shown that reasonable service candidates with various granularity are successfully extracted from the source code.

2. Preliminaries

2.1. Service-Oriented Architecture (SOA)

The service-oriented architecture (SOA) is a software architecture that regards software functionalities as services (which we call SOA services), and builds a system by integrating and orchestrating the multiple services. Although there are various definitions, in this paper we define an SOA service as a set of processes (procedures) satisfying the following three conditions S1, S2, and S3.

(Condition S1) Open Interface: A service has an open in-
terface, by which external entities can access to the service independently of the implementation of the service. For the access, a service cannot require platform specific operations, nor implementation-specific data that are only used within the system.

(Condition S2) Self-Contained: A service can be executed by itself without any other services. Thus, a process cannot be a service if the process requires execution and/or data of any other processes. Such mutually-dependent processes should be aggregated within the same service.

(Condition S3) Coarse-Grained: A service is a coarse-grained process that can be a business construct by itself. Also, multiple services can be integrated to achieve a more sophisticated and coarser-grained service.

The above conditions are necessary conditions for SOA services, and contribute to the loose coupling among services [8][11]. Thus, the services can be easily composed and decomposed to implement various business workflows. As a result, the SOA can make a system robust and flexible for rapid changes of business environment.

2.2. SOA Legacy Migration

The SOA legacy migration refers to a re-engineering activity that converts the legacy system to an SOA-enabled system. In the conventional SOA development frameworks (e.g., [2][3][5]), the services are usually identified at the business modeling and analysis phases. Every business process is modeled and refined into elementary processes that cannot be decomposed further. Each elementary process corresponds to an atomic service, associated with a software module implemented by fine-grained components or libraries. Although the services are optimally determined in a top-down manner, there is no guarantee that the legacy system implements modules that exactly correspond to the services. Adapting and refactoring the legacy system to the optimal services usually requires huge cost.

It is thus reasonable in the SOA legacy migration to adopt a bottom-up approach, which starts with the system analysis so that the current implementation is reused as much as possible. To tackle this, there have been several relevant studies [6][9][10][12]. As mentioned in these studies, a major challenge lies in how to identify services in the legacy system. We will review these studies in Section 5.3, compared with the proposed method.

2.3. The Service Extraction Problem

To support the SOA legacy migration effectively, we tackle the following problem in this paper.

Input: Source code $C$ of a legacy system. We assume that $C$ is written in a procedural program language.

Output: A set of services $S = \{ s_1, s_2, ..., s_n \}$, where every $s_i$ is an aggregation of processes (procedures) within $C$ that satisfies Conditions S1, S2 and S3.

2.4. Liquor Shop Inventory Control System

To help understanding, we introduce a liquor shop inventory control system, as an illustrative example of a legacy system. The system is an implementation of the “Liquor Shop Problem (Sakaya-Mondai)”, which is a common problem in the software engineering education in Japan [13]. The following actors appear in the problem.

Customer orders products to the liquor shop.

Stock Manager manages the inventory of the liquor shop, and executes business processes like “Ship Products”, “Receive Products”, “Resolve Out of Stock”.

Freighter ships products to the customer, and also delivers products to the warehouse.

Warehouseman handles input/output of the warehouse based on the instruction from the stock manager.

In the business processes, various documents, such as “Order Form”, “Bin (Container) Manifest”, “Shipping Instruction” and “Out of Stock Notice” are exchanged among the actors.

Figure 1 shows an implementation of the “Resolve Out of Stock” process, written in the C language. The business process is explained in terms of comments in the code. This program is extensively used in the following discussion.
3. Proposed Method

3.1. Introducing DFD

To achieve the service extraction from source code, we extensively use the Data From Diagram (DFD) [7]. The DFD is a diagram visualizing processes in a system as well as data flows among processes. It has been well accepted in the structured analysis of a system. In a DFD, an oval represents a process, a solid arrow represents a data flow, a pair of parallel lines represents a data store, and a box represents an external entity. Figure 2 shows an example of the DFD, corresponding to the source code in Figure 1.

The reasons why we chose the DFD as a tool are as follows. First, the DFD is well-suited to legacy systems, since they are often written in the procedural structured language. Second, since the DFD visualizes processes (not objects), it helps us to find services (= processes), intuitively. Third, the DFD can describe multiple layers to represent different abstraction levels. So it allows us to investigate services with various granularities.

3.2. Key Ideas for Service Extraction

As defined in Section 2.1, every SOA service is a process in a system. However, every process in a system is not necessarily an SOA service. So we evaluate processes in the DFD according to Conditions S1, S2 and S3.

3.2.1 Condition S1: Analyzing Open Interface

A process in a DFD corresponds to a procedure (or function) of source code. Data flows to/from the process characterize input/output interfaces of the procedure. In order for a procedure to be a service with an open interface, the input/output data must be common enough for service consumers to understand. We measure such commonality as the degree of how widely the data is known within the system. If data is exhibited to external actors or shared by many processes, we consider that the data is common. On the other hand, if data is exchanged only by a few limited processes, we regard the data is not common.

Our key idea is to evaluate the degree of open interface as the commonality of the input/output data of the process. To do this, we classify every data flow in the DFD into three categories: (1) external data – data exchanged with external actors, (2) system data – data accessed commonly from various processes (e.g., database, global variables, etc.) (3) module data – data used by limited processes only (e.g., local variables, temporal data, etc.)

1We may also use the term process to represent a procedure or a task in the procedural program.

3.2.2 Condition S2: Analyzing Self-Containedness

A service can be executed by itself without depending on other services. Therefore, two processes that have strong dependency cannot be two separate services, and they should be aggregated within the same service. We analyze such dependency among processes from the viewpoints of data and control.

The data dependency is caused by data exchanged among the processes. Using the data category marked in the DFD, we identify two kinds of data dependency: (MD) module data dependency and (SD) system data dependency. We consider that processes exchanging uncommon data has strong dependency, since no other process can directly interpret the uncommon data.

The control dependency is caused by control flow among between processes. Using the DFD and the source code, we identify two kinds: (TR) transaction dependency and (CO) condition dependency. The transaction dependency is strong dependency such that all processes must be executed together in the same transaction. The condition dependency is relatively weak dependency, where a process specifies a condition for execution of other processes.

3.2.3 Condition S3: Coarse-Granularity for Business

Processes with different granularity appear in different layers of the DFD. Therefore, by investigating each level of the (hierarchical) DFD, a user of the proposed method can extract service candidates with various granularity. Thus, the user can choose appropriate granularity level for the target system and business goal.

3.2.4 Service Extraction Rules

Even if a process does not satisfy Condition S1 or S2, the process can become a service when combined with other processes. We present six rules for the service extraction, which systematically aggregate mutually-dependent processes.
3.3. Outline of Service Extraction

The proposed method for the service extraction problem (in Section 2.3) consists of the following four steps.

(STEP1) Obtain a hierarchical DFD from C.
(STEP2) Categorize data flows in the DFD.
(STEP3) Analyze dependency among processes.
(STEP4) Apply the service extraction rules.

Due to the limited pages, we do not discuss details of Step 1 in this paper. The conventional reverse-engineering techniques [1] [4], which derive the hierarchical DFD from procedural programs, can be used for Step 1.

3.4. Categorizing Data Flows (STEP2)

STEP2 classifies data flows in the DFD into the three categories mentioned in Section 3.2.1. For the DFD shown in Figure 2, Figure 3 shows a resultant DFD obtained in STEP2. We explain each data category as follows.

3.4.1 External Data (E)

We define the external data as data exchanged between a process and an external actor. In the actual system, files, standard input/output and printed documents are typical instances. In Figure 3, “Shipping Instruction” is an external data. In the DFD, we label “E” to represent the external data flows.

3.4.2 System Data (S)

The system data is data commonly used by many processes in the system. Typical instances are input/output for database, global variables, shared data among sub-systems. In the DFD, data store shared by multiple processes can be system data. We label “S” in the DFD to represent the system data flows. In Figure 3, “Out of Stock Notice” is shared by five processes, so it can be system data. Data for two DBs (“OoSN DB”, “Liquor Shop Inventory DB”) are too.

3.4.3 Module Data (M)

The module data is specific data used by a few limited processes. Typical instances are temporal variables and local variables. In the DFD, a direct data flow between two processes, or a data store shared with limited processes only can be classified as module data. We label “M” in the DFD to represent the module data flows. In Figure 3, “# of Bins” is obtained by process “(3) Reserve Inventory”, and used for “(4) Print SI Header” only. So we make it module data. Similarly, “List of Picking Bins” is module data since it is temporal data used for “(5) Print SI Data” only.

3.5. Analyzing Dependency (STEP3)

Using the result of STEP2 and the source code, STEP3 analyzes the dependency between processes, with respect to the data dependency (MD, SD) and the control dependency (TR, CO). Here, we consider MD and TR to be strong dependency, whereas SD and CO are regarded as to be weak dependency. The strong dependency takes precedence over the weak one when multiple relations hold simultaneously.

For convenience, the data dependency is shown in a dotted arrow(→) in the DFD, while the control dependency appears as an alternate long and short dashed arrow(→). In the following, let \( P_1, P_2 \) be arbitrary processes, \( d \) be any data. Also, we write \( L(d)(\in \{M, S, E\}) \) to represent the data category of \( d \) (defined in STEP2).

3.5.1 Module Data Dependency (MD)

We say that processes that exchange module data have module data dependency. By definition, the module data is so uncommon (specific) that it cannot be produced or consumed easily by external actors or other processes. Hence, we consider that processes exchanging the module data have strong dependency, and they are tightly coupled.

Now we write \( P_1 \xrightarrow{d} P_2 \) to represent a data flow \( d \) from \( P_1 \) to \( P_2 \) (including an indirect flow via a data store). Then the module data dependency from \( P_1 \) to \( P_2 \), denoted by \( MD(P_1, P_2) \), is defined as follows:

\[
MD(P_1, P_2) \iff \exists d : [(L(d) = M) \land (P_1 \xrightarrow{d} P_2)]
\]

As for the example in Figure 3, we can identify \( MD((3), (4)) \) and \( MD((3), (5)) \). In the DFD, the data dependency is labeled by “MD”.

3.5.2 System Data Dependency (SD)

We say that processes that share system data have system data dependency. By definition, the system data is common
and opened to many processes. Therefore, we consider that
the system data dependency is weaker than the module data
dependency. The system data dependency from P1 to P2,
denoted by SD(P1, P2), is defined as follows:

\[ SD(P1, P2) \iff \exists d : \left[ (\ell(d) = S) \land (\neg MD(P1, P2)) \land \left( P1 \rightarrow P2 \right) \right] \]

As for the example in Figure 3, we can identify SD((3), (2)) via “Liquor Shop Inventory DB”, SD((6), (7)) via “OoSN DB”, and so on. In the DFD, the system data dependency is labeled by “SD”.

Figure 4 shows the DFD showing the the data dependency on the DFD in Figure 3.

### 3.5.3 Transaction Dependency (TR)

We say that processes that must be executed in the same transaction have transaction dependency. The transaction is a process control where multiple processes are executed at once in a consistent manner. We write TR(P1, P2) to represent the transaction dependency between P1 and P2. Typical cases of TR(P1, P2) include (a) P1 must be executed before P2, or (b) executing both P1 and P2 completes a task (i.e., omitting one of them produces an incomplete result). In the source code, we often identify such transaction dependency within processes in the same code block.

Let us take the source code in Figure 1 and the DFD in Figure 3. For instance, we can identify TR((4), (5)), since a complete shipping instruction requires both header and data body. Any pair of processes (3), (4), (5), (6) has transaction dependency, since all of them should be performed in the same transaction as specified in the same code block. The transaction dependency is labeled by “TR” in the DFD.

### 3.5.4 Control Dependency (CO)

If execution of P2 depends on a condition evaluated by P1 (i.e., P1 works as a control flag of P2), we say that P1 and P2 have control dependency. Let IF(P1, P2) be a predicate that P1 is a control flag of P2. Then the control dependency, denoted by CO(P1, P2), is defined as follows, taking the priority against TR.

\[ CO(P1, P2) \iff (\neg TR(P1, P2)) \land (IF(P1, P2)) \]

In a situation where CO(P1, P2), P1 just describes a context under which P2 is executed. Altering P1, P2 may be executed by other contexts. We thus consider the control dependency is weaker than the transaction dependency. In the DFD, the control dependency is labeled by “CO”.

As for the example of Figures 1 and 3, processes (1) and (2) respectively specify the context of execution of (3), (4), (5) and (6). So we identify the control dependency.

Figure 5 shows the DFD showing the the control dependency on the DFD in Figure 3. To avoid the schematic complexity, we make a group of (3)-(6) with the transaction dependency, and draw an arrow with CO from (1) (or (2)) to any in the group.

### 3.6. Extracting Services (STEP4)

Using the dependency obtained in STEP3, this step aggregates mutually-dependent processes, and extracts them as self-contained services with open-interface.

Suppose that certain dependency is identified between P1 and P2 in STEP3. If P1 and P2 are aggregated within the same service, we call the aggregation an integrated process, denoted by P1 + P2. While, if P1 and P2 can be separated services, we call them separated processes, denoted by P1/P2. Here we present six rules of the service extraction that systematically integrate or separate the processes.

**(Rule1) Integrate Processes with MD**

Processes P1 and P2 such that MD(P1, P2) should be aggregated within the same service. If they are separated services, the service consumer has to bridge the module data
Suppose that we have two integrated processes:

(Rule 5) Integrate Merged Processes

for many processes. If the data store between $P_1$ and $P_2$ stores the system data appropriately, we consider that either $P_1$ or $P_2$ can be executed asynchronously. For this, the input/output data is reasonably common for the service consumer (calling processes). Thus, we consider that both $P_1$ and $P_2$ satisfy Conditions S1 and S2. Thus, for $P_1$ and $P_2$ such that $SD(P_1, P_2)$, we can make $P_1 | P_2$. Note that the separation is not mandatory. We can integrate $P_1 + P_2$ if necessary. If $TR(P_1, P_2)$ holds simultaneously, the following Rule 3 should be applied first. In Figure 4, this rule separates $(6) | (1), (3) | (2)$, etc.

(Rule 2) Separate Processes with SD

Processes $P_1$ and $P_2$ such that $SD(P_1, P_2)$ can be separated as different services. The system data is common enough for many processes. If the data store between $P_1$ and $P_2$ stores the system data between $P_1$ and $P_2$ such that $MD(P_1, P_2)$, we make $P_1 + P_2$. In Figure 4, this rule aggregates $(3) + [(4) + (3) + [(5)$.

(Rule 3) Integrate Processes with TR

Processes $P_1$ and $P_2$ such that $TR(P_1, P_2)$ should be aggregated within the same service. Since $P_2$ presupposes $P_1$, $P_2$ cannot be executed by itself. If $P_1$ and $P_2$ are separated services, the service consumer must consider the execution order and transaction of the two services, which violates Condition S2. Thus, for $P_1$ and $P_2$ such that $TR(P_1, P_2)$, we make $P_1 + P_2$. In Figure 5, this rule aggregates $(3) + [(4), (3) + [(5), (3) + [(6), (4) + [(5), (4) + [(6), (5) + [(6)$.

(Rule 4) Separate Processes with CO

Processes $P_1$ and $P_2$ such that $CO(P_1, P_2)$ can be separated as different services. $P_1$ just specifies the context of $P_2$. So we consider it reasonable to execute $P_2$ under another context, by altering $P_1$ with another process. Of course in this case, $P_2$ must be implemented without having module data dependency with $P_1$. Thus, for $P_1$ and $P_2$ such that $CO(P_1, P_2)$, we can make $P_1 | P_2$. Note that the separation is not mandatory. If $MD(P_1, P_2)$ holds simultaneously, Rule 1 is applied first. Figure 5, this rule separates $(1) | (3), (2) | (3)$, etc.

(Rule 5) Integrate Merged Processes

Suppose that we have two integrated processes: $P_1 + P_2$ and $P_2 + P_3$. Then, executing $P_2$ requires both $P_1$ and $P_3$. Therefore, we need to integrate $P_1$, $P_2$ and $P_3$ into $P_1 + P_2 + P_3$. This rule applies to processes with MD or TR. In Figure 5, this rule makes $(4) + [(5) + [(6)$ from $(4) + [(6)$ (obtained by Rule 3), and $(5) + [(6)$ (obtained by Rule 3), etc.

(Rule 6) Merge Transitive Processes

Suppose that we have two integrated processes: $P_1 + P_2$ and $P_2 + P_3$. Then, executing $P_3$ requires $P_2$, and also executing $P_2$ requires $P_1$. Therefore, we need to integrate $P_1$, $P_2$ and $P_3$ into $P_1 + P_2 + P_3$. This rule applies to processes with MD or TR. In Figures 4 and 5, this rule makes $(3) + [(4) + [(5) + [(6)$ from $(3) + [(4)$ (obtained by Rule 1) and $(4) + [(5) + [(6)$ (obtained by Rule 5).

Figure 6 shows services extracted from the “Resolve Out of Stock” process. In this example, four service candidates were derived: “Check Pending OoSN Service”, “Check Inventory Service”, “Resolve OoSN Service”, and “OoSN Garbage Collection Service”. For instance, “Resolve OoSN Service” is such a service that inputs an out of stock notification (OoSN), reserves inventory, creates a shipping instruction, and updates the OoSN database as the transaction is done. It can be seen that (1) each of the four services can be executed by itself (i.e., self-contained), and that (2) the service has interfaces with commonality of the system data.

4. Case Study

We have conducted a case study of the service extraction from one implementation of the liquor shop inventory control system. This implementation is written in the C language, comprising about 800 lines of code. By reverse-engineering, we obtained a hierarchical DFD from the implementation. Then, we applied the proposed method to the top 3 layers (Layer 0 (= Context Diagram), Layer 1 and Layer 2) of the DFD.

Figure 7 shows the services extracted from the DFD Layer 1, describing sub-systems of the whole Liquor Shop...
Figure 7. Services extraction from upper layer

System. From this layer, we derived five services. Although there are dependency $SD((2), (3))$, $SD((1), (4))$ and $SD((1), (3))$, these processes can be separated according to Rule 2. Since we have no other rules applicable, we extract (1)|2|3|4|5 as five services in this layer. The five services are “Create OoSN DB Service”, “Receive Service”, “Ship Service”, “Resolve Out of Stock Service”, “Delete Empty Bins Service”, all of which are well suited to our intuition of business service. It can be seen from the result that this implementation was well structured, in accordance with the original business processes of the Liquor Shop Problem.

Table 1 shows all services extracted from different layers of the DFD. In the table, the column layer represents the layer of the DFD, where 0 corresponds to the top level DFD (= context diagram), 1 corresponds to the one in Figure 7, and 2 corresponds to DFDs which refine five processes in Figure 7 (e.g., the DFD in Figure 2). As seen in the table, we can see that all extracted services are reasonable and consistent for the Liquor Shop Problem. Although the granularity varies, we have confirmed that every service can be executed by itself and take input/output as common as system data.

5. Discussion

5.1. Evaluation of Extracted Services

5.1.1 Hiding Implementation-Specific Logic

The proposed method derives services that encapsulates control flows of internal processes as well as implementation-specific data. According to the service extraction rules, processes that requires a specific execution order or transaction are aggregated within the same service. So service consumers do not need to care for the control flow among the processes. Also the module data, which often depends on the implementation, is encapsulated and never appears in the service interface.

The extracted service requires the system data or external data for the input/output. For this, the system data may not always be exhibited directly in the service interface of SOA (e.g., WSDL of Web service), since the system data may be represented in an implementation-specific form to increase the performance of data sharing. For such a case, we assume to apply a service wrapper [12] that simply converts the system data into an implementation-neutral form.

By the above discussion, every service obtained by the proposed method have open interface with common data, which satisfies Condition S1. Also, every service can be executed by itself, without considering the execution of other services or processes, which satisfies Condition S2.

5.1.2 Extracting Multi-Granularity Services

The proposed method can be applied to any layer of the hierarchical DFD. If applied to a higher layer, we can extract coarser-grained services with high-level and sophisticated functionalities. On the other hand, when applied to a lower layer, we can expect finer-grained services, which have low-level but re-usable services. Thus, by choosing an appropriate layer optimal for the target business and application, the user can extract services with appropriate granularity, which can satisfy Condition S3.
5.2. Limitations of Proposed Method

The proposed method tries to find service candidates by investigating structured processes within the DFD. Therefore, services can be extracted successfully from well-structured source code, such as the one used in our case study. However, from seriously ill-structured source code, there is no guarantee to be able to obtain services with appropriate granularity. For such a case, some refactoring process would be necessary before the service extraction.

Another limitation is that the data classification in STEP2 relies on the human expertise. Especially distinction of module data and system data would be difficult in some cases. For instance, a programmer may implement certain module data as a global variable just for convenience. There is also a question that data shared by four modules should be system data or module data. Currently all the decision of the data classification is left to the user of the proposed method, which may influence the service extraction result.

To overcome the above limitation, we plan to investigate a program refactoring method, which counts the degree of ease of the SOA migration for the given program structure. Also, it is important in our future work to define a quantitative metrics that evaluates the commonality of data.

5.3. Related Work

Lewis et al. [9] developed a software process called SMART, which provides preliminary analysis of feasibility, strategy, cost and risk for the legacy migration to the SOA. Cetin et al. [6] presented a migration approach based on the service mash-up. These are total frameworks of migration, where each step of the migration must be implemented by concrete methods. The proposed method can contribute to implementing these frameworks, especially in analyzing the legacy system for identifying the existing reusable services.

Sneed [12] proposed a method that salvages and wraps the legacy source code. In this method, the analyst identifies business rules (i.e., services) at the source code level, conducting data flow analysis focusing interesting variables. However, the method does not especially count the characteristics of the SOA services. Thus, the derived services may vary depending on the expertise of the analyst. Our method takes Conditions S1-S3 explicitly, which enables consistent and objective service extraction.

Matos et al. [10] presented a migration method based on code graphs obtained from annotated source code. As the authors mentioned, a big challenge lies in the functional code annotation process identifying potential services within the source code. For this, they presented a couple of useful code patterns, but the consolidation of the code patterns is left to future work. Our method provide concrete rules and procedures of the service extraction, although the applications is limited to the procedural programs only.

6. Conclusion

In this paper, we have presented a method that extracts SOA services from the procedural program and its data flow diagram (DFD), by analyzing data and control dependency among processes. A case study with a liquor shop inventory control system showed that the proposed method can derive reasonable consistent services with various granularity. Our future work includes; the refactoring method for efficient SOA migration, systematic data classification, and evaluation metrics presented in Section 5.2.

Acknowledgments This research was partially supported by the Japan Ministry of Education, Science, Sports, and Culture, Grant-in-Aid for Young Scientists (B) (No.21700077, No.20700027).

References