Identifying Services in Procedural Programs for Migrating Legacy System to Service Oriented Architecture

Masahide Nakamur, Kobe University, Japan

Hiroshi Igaki, Tokyo University of Technology, Japan

Takahiro Kimura, Nihon Unisys Ltd., Japan

Ken'ichi Matsumoto, NAIST, Japan

ABSTRACT

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

Keywords: Data Flow Diagrams, Dependency Analysis, Legacy Migration, Procedural Program, Service Extraction, Service Oriented Architecture, Source Code

INTRODUCTION

Due to rapid changes in business environment, enterprise software systems are required to be more agile and flexible to keep up with the changes. However, most enterprise systems have been built upon 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)* (Erl, 2007; Newcomer & Lomow, 2004; Papazoglow & Georgakopoulos, 2003) is an architecture paradigm to cope with the problem.

DOI: 10.4018/jisss.2011100104

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 or 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 (Cetin, Altintas, Oguztuzun, Dogru, Tufekci, & Suloglu, 2007; Lewis, Morris, Smith, & Simanta, 2008). Most of the conventional SOA development frameworks e.g., SOMA (Arsanjani, Ghosh, Allam, Abdollah, Gariapathy, & Holley, 2008), SOMF (Bell, 2008), and BMM (Berkem, 2008) adopt a top-down approach, which starts with the business process analysis, identifies elementary processes, and implements them as services. Since the system is designed optimally for the SOA, the top-down approach is well applied to development of brand-new systems. However, it does not consider much how to reuse the existing legacy system.

To support the SOA legacy migration effectively, this paper presents a pragmatic method that extracts candidates of SOA services from procedural programs. In the proposed method, we extensively analyze *dependencies* among processes (i.e., procedures) in the source code. Each service is derived as an aggregation of mutually-dependent processes, so that the service has open-interface, self-containedness and coarse granularity for the business.

For implementing the method, we first obtain the *Data Flow Diagram (DFD)* (DeMarco, 1979) by applying reverse-engineering techniques to the given source code (O'Hare & Troan, 1994; Benedusi, Cimitile, & Carlini, 1989). We then classify every data flow in the DFD into three categories (*external, system* and *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.

To evaluate the proposed method, we have conducted two kinds of experiments with a liquor shop inventory control system. The experimental results show that reasonable service candidates with various granularities are successfully extracted from the source code. We also investigate the derived services through the comparison with the classical software metrics: *cohesion* and *coupling* metrics (Al-Ghamdi, Shafique, Al-Nasser, & Al-Zubaidi, 2001; Lakhotia, 1993; Yourdon & Constantine, 1979).

The paper is organized as follows. In the next section, we briefly survey the service oriented architecture and the problem to be tackled. We then present the proposed service extraction method. We conduct the experiment with the inventory control. Next, we evaluate the proposed method with several related work and finally, we conclude the paper.

PRELIMINARIES

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 interface*, 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, or 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 (Erl, 2007; Newcomer et al., 2004). 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.

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., SOMA (Arsanjani et al., 2008), SOMF (Bell, 2008), and BMM (Berkem, 2008), 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 (Cetin et al., 2007; Lewis et al., 2008; Matos & Heckel, 2009; Sneed, 2006). As mentioned in these studies, a major challenge lies in *how to identify services in the legacy system*. We will review these studies later on.

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.

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 (Yamazaki, 1984). 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”.

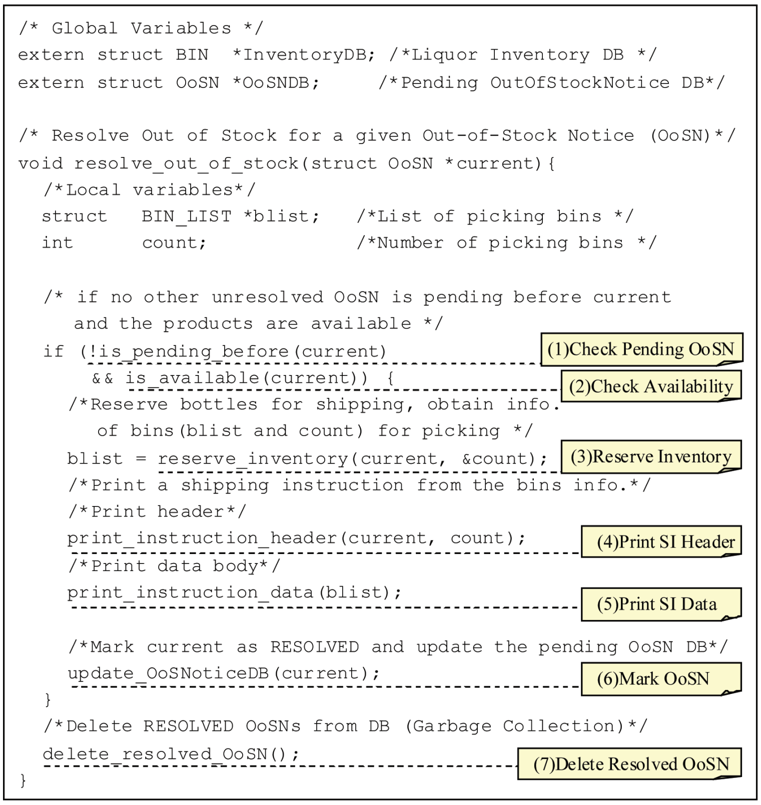
[Freighter] ships products to the customer, and also delivers products to the warehouse.

[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”, “Container (BIN) 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.

Figure 1. Source code of ``Resolve Out of Stock" process



```
/* Global Variables */
extern struct BIN  *InventoryDB; /*Liquor Inventory DB */
extern struct OoSN *OoSNDB;     /*Pending OutOfStockNotice DB*/

/* Resolve Out of Stock for a given Out-of-Stock Notice (OoSN)*/
void resolve_out_of_stock(struct OoSN *current){
  /*Local variables*/
  struct   BIN_LIST *blist;   /*List of picking bins */
  int      count;             /*Number of picking bins */

  /* if no other unresolved OoSN is pending before current
     and the products are available */
  if (!is_pending_before(current)
      && is_available(current)) {
    /*Reserve bottles for shipping, obtain info.
       of bins(blist and count) for picking */
    blist = reserve_inventory(current, &count);
    /*Print a shipping instruction from the bins info.*/
    /*Print header*/
    print_instruction_header(current, count);
    /*Print data body*/
    print_instruction_data(blist);

    /*Mark current as RESOLVED and update the pending OoSN DB*/
    update_OoSNoticeDB(current);
  }
  /*Delete RESOLVED OoSNs from DB (Garbage Collection)*/
  delete_resolved_OoSN();
}
```

The business process is explained as follows:

[Trigger Condition] The process is triggered when new stock arrives. This implementation takes an Out-Of-Stock Notice (say, *current*) as input. Each Out-of-Stock Notice (OoSN) describes an order (liquor brand and quantity) that had not been processed due to out of stock.

[Resolution Policy] The given OoSN (*current*) can be *resolved* only if [(a) there exist no other pending OoSN issued before *current* and requesting the same brand as *current* does], and [(b) there exists sufficient stock in the inventory].

[Shipping Instruction] When *current* is resolved, the requested quantity of the product is reserved from the inventory. Then, a Shipping Instruction (SI) is issued. Every SI describes a list of pairs [Bin No. and Qty], telling how many bottles should be picked from which bins.

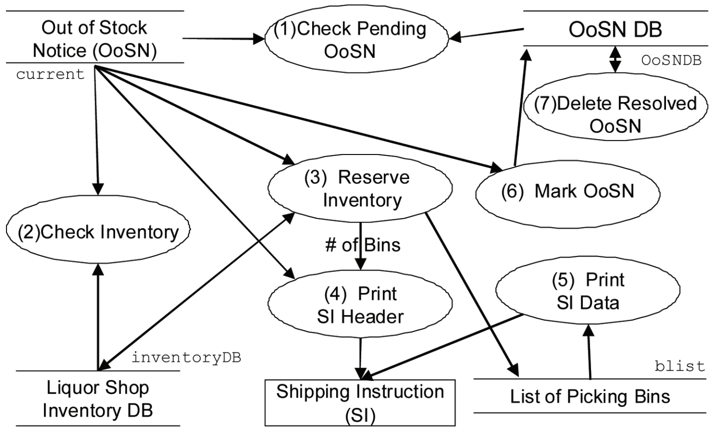
[Delete of OoSN] The resolved OoSN is deleted from the database after the shipping.

PROPOSED METHOD

Introducing DFD

To achieve the service extraction from source code, we extensively use the *Data Flow Diagram* (*DFD*) (DeMarco, 1979). 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* (we may also use the term process to represent a

Figure 2. DFD of "Resolve Out of Stock" process



procedure or a *task* in the procedural program), 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.

Key Ideas for Service Extraction

As defined before, 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.

[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 that 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.)

[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 have 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.

[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 granularities. Thus, the user can choose appropriate granularity level for the target system and business goal.

[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.

Outline of Service Extraction

The proposed method for the service extraction problem 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.

Obtaining a Hierarchical DFD (STEP1)

We first obtain a *hierarchical DFD* from a given C program. As defined in DeMarco (1979), a hierarchical DFD consists of multiple *layers*, each of which contains processes at a certain level of abstraction, and data flows among the processes. A process can be *expanded* to show a more detailed DFD in a lower layer. Thus, we can see processes with different *granularity* at different layers of the DFD. That is, a higher layer contains coarse-grained processes implementing high-level functionalities, whereas a lower layer includes fine-grained processes performing more primitive functionalities.

In this paper, we do not discuss the technical details on how to implement STEP1. The conventional reverse-engineering techniques (O'Hare et al., 1994; Benedusi et al., 1989), which derive the hierarchical DFD from procedural programs, can be used. In the reverse-engineering, the layers are usually derived based on a function call graph (or a structure chart), and the granularity is determined how each function is modularized in the original source code.

The following steps are supposed to be performed for *every* layer of the hierarchical DFD.

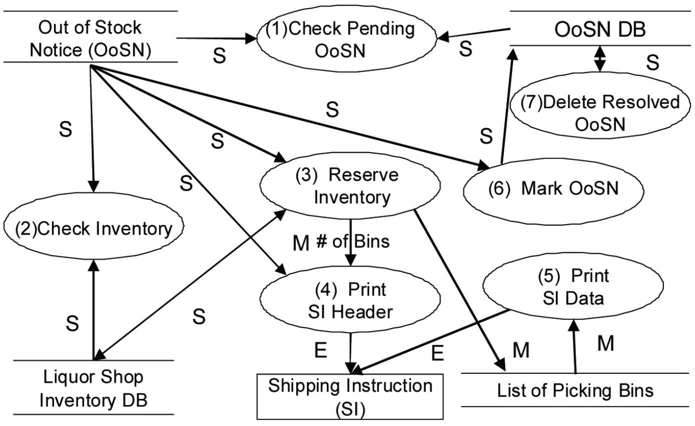
Categorizing Data Flows (STEP2)

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

[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.

[System Data (S)] The system data is data commonly used by processes in the system.

Figure 3. DFD after data classification



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. So are the data for two DBs ("OoSN DB", "Liquor Shop Inventory DB").

[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" 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.

We are currently assuming that STEP2 should be supported by human expertise of system maintainers. Labeling "E" is quite easy since the data is connected to external entity. However, deciding "S" or "M" is sometimes not obvious, since it needs to evaluate the commonality of data. One criterion is to count the number of processes related to the data. However, the final decision should be made, considering the semantics and the roles of the data. These are not described in the source code syntactically, but are in the knowledge of the system maintainers. This topic will be discussed later.

Analyzing Data/Control Dependencies (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).

[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 interdependency, and they are tightly coupled.

Now we write $P_1 - (d) \rightarrow P_2$ 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:

$MD(P_1, P_2) \Leftrightarrow \exists d$: $[(L(d) = M) \wedge (P_1 - (d) \rightarrow P_2)]$

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

[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 P_1 to P_2, denoted by $SD(P_1, P_2)$, is defined as follows:

$SD(P_1, P_2) \Leftrightarrow \exists d$: $[(L(d) = S) \wedge (\neg MD(P_1, P_2)) \wedge (P_1 - (d) \rightarrow P_2)]$

In Figure 3, we can see *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 data dependency on the DFD in Figure 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 $TR(P_1, P_2)$ to represent the transaction dependency between P_1 and P_2. Typical cases of $TR(P_1, P_2)$ include (a) P_1 must be executed before P_2, or (b) executing both P_1 and P_2 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.

[Condition Dependency (CO)] If execution of P_2 depends on a condition evaluated by P_1 (i.e., P_1 works as a control flag of P_2), we say that P_1 and P_2 have *condition dependency*. Let $IF(P_1, P_2)$ be a predicate that P_1 is a control flag of P_2. Then the control dependency, denoted by $CO(P_1, P_2)$, is defined as follows, taking the priority against *TR*.

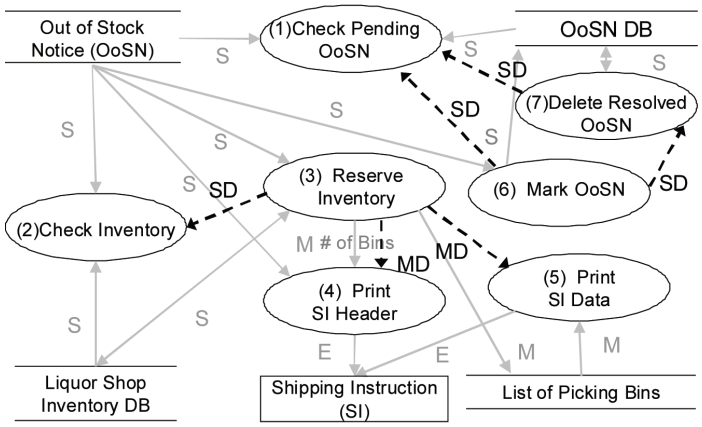
$CO(P_1, P_2) \Leftrightarrow \neg \mathrm{TR}(P_1, P_2) \wedge \mathrm{IF}(P_1, P_2)$

In a situation where $CO(P_1, P_2)$, P_1 just describes a *context* under which P_2 is executed. By altering P_1, P_2 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 Figure 1 and Figure 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 control dependency on the DFD in Figure 3. To avoid the schematic complexity, we make a group of (3)-(6) with the transaction depen-

Figure 4. DFD after data dependency analysis



dency, and draw an arrow with CO from (1) (or (2)) to the group. This is to abbreviate arrows from (1) (or (2)) to any in the group.

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.

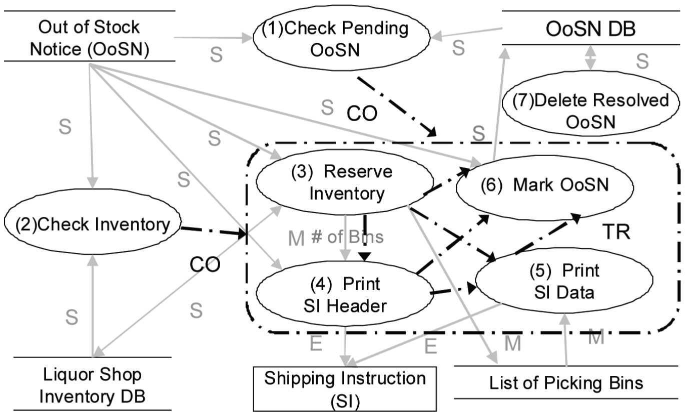
Suppose that certain dependency is identified between P_1 and P_2 in STEP3. If P_1 and P_2 are aggregated within the same service, we call the aggregation an *integrated process*, and represent it by P_1 +[P_2. If P_1 and P_2 can be separated services, we call them *separated processes*, and represent them by $P_1 \mid P_2$. Here we present six rules of the service extraction that systematically integrate or separate the processes.

[(Rule1) Integrate Processes with MD] Processes P_1 and P_2 such that $MD(P_1, P_2)$ should be aggregated within the same service. If they are separated services, the service consumer has to *bridge* the module data between P_1 and P_2, by executing P_1 and P_2 in order. This is against Condition S2. Also, the module data is uncommon data, which is against Condition S1. Thus, for 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) and (3)+[(5).

[(Rule2) 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 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 $P_1 \mid P_2$. 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 makes separated processes like (6)|(1), (3)|(2), etc.

[(Rule3) 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

Figure 5. DFD after control dependency analysis



aggregates (3)+[(4), (3)+[(5), (3)+[(6), (4)+[(5), (4)+[(6), (5)+[(6).

[(Rule4) 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 $P_1 \mid P_2$. 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.

[(Rule5) Integrate Merged Processes] Suppose that we have two integrated processes: P_1 +[P_3 and P_2 +[P_3. Then, executing P_3 requires both P_1 and P_2. 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 Rule3), and (5)+[(6) (obtained by Rule 3), etc.

[(Rule6) Integrate 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 Figure 4 and Figure 5, this rule makes (3)+[(4)+[(5)+[(6) from (3)+[(4) (obtained by Rule 1) and (4)+[(5)+[(6) (obtained by Rule 5).

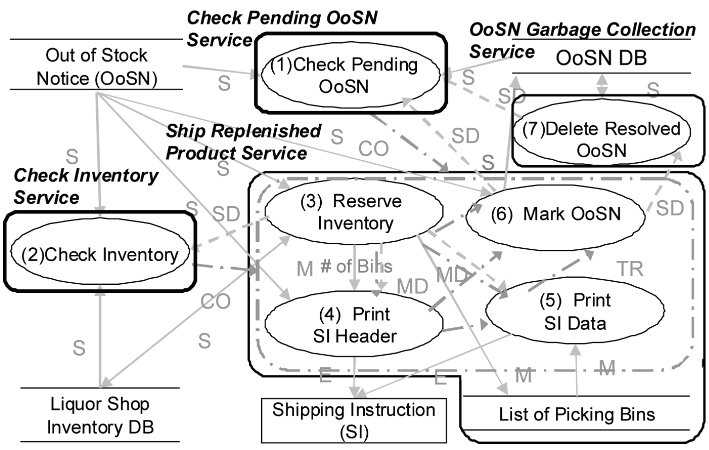
Figure 6 shows services extracted from the "Resolve Out of Stock" process.

In this example, the following four service candidates were derived:

1. **Check Pending OoSN Service:** For a given *x* of Out of Stock Notice (OoSN), check if there is any other pending OoSN requesting the same product as *x*'s but issued earlier than *x*.
2. **Check Inventory Service:** For a given x of OoSN, check the availability of the product requested by *x* within the current inventory.
3. **Ship Replenished Product Service:** Ship the requested product as the product is supposed to be replenished. The service reserves the inventory, creates a shipping instruction, and updates the OoSN database as the transaction is done.
4. **OoSN Garbage Collection Service:** Delete the "resolved" OoSN from the data base.

It can be seen that (1) each of the four services can be executed by itself (i.e., self-

Figure 6. Service candidates within ``Resolve Out of Stock" process



contained), and that (2) the service has interfaces with commonality of the system data.

EXPERIMENT

To demonstrate the effectiveness of the proposed method, this section conducts two kinds of experiments of the service extraction.

Experiment 1: Service Extraction from Different Implementation

The first experiment is to extract services within the "Resolve Out of Stock" process from *another* implementation. The aim of the experiment is to see how the proposed method is adapted to various design choices for the same requirement.

Figure 7 shows another implementation of the "Resolve Out of Stock" process. The source code has been written by a different programmer based on the same specification. Although the workflow seems to be similar to the previous program (Figure 1), the following points are different in details.

Point 1: The process is performed for a given cargo manifest (not an OoSN). This is because the replenishment of the out-of-stock product is possible only when a cargo arrives.

Point 2: For every cargo manifest, all of the OoSNs are scanned to be resolved.

Point 3: The shipping is performed when the cargo manifest contains the requested product.

Point 4: Both "checking inventory" and "checking the pending OoSNs" are performed within a single function is_replenished().

Figure 8 (a), (b), (c) show the resulting DFDs obtained after STEP 2, 3, 4, respectively.

From this implementation, the following four services have been identified:

1. **Check Cargo Manifest Service:** For given *cm* of a cargo manifest and x of an Out of Stock Notice (OoSN), check if *cm* contains products requested by *x*.
2. **Check Replenishment Service:** For a given *x* of OoSN, check if the product requested by x can be replenished. Specifically, check if there exists no other pending OoSN requesting the same product as *x*'s but issued earlier than *x*. Also, check the availability of the product within the current inventory.
3. **Ship Replenished Product Service:** Ship the requested product as the product is supposed to be replenished. The service reserves

the inventory, creates a shipping instruction, and updates the inventory database.

4. **Update Resolved OoSN Service:** Delete the "resolved" OoSN from the data base.

Now we compare the services with the ones in the previous example (Figure 6). First, (A') Check Cargo Manifest Service is a completely new service which does not exist in the previous example. Second, (B') Check Replenishment Service is a coarse service that involves functionalities of both (A) Check Pending OoSN Service and (B) Check Inventory Service. Next, (C') Ship Replenished Product Service is almost the same as (C) Ship Replenished Product Service. Finally, (D') Update Resolved OoSN Service is the same as (D) OoSN Garbage Collection Service.

In this experiment, it can be seen that similar but slightly different services have been identified from different implementations, even though the implementations realize the same business process. Each of the obtained services reflects well the design choice considered in the implementation. Thus, the proposed method derives the AS-IS services, making full re-use of the existing legacy code.

The result of service extraction can be used to evaluate the current implementation with respect to the *degree of ease* of legacy migration. If no reasonable service is extracted from the source code, it means that major revision for *untangling* tightly-coupled modules will be required, resulting in a huge migration effort.

Experiment 2: Extracting Multi-Grained Services

The second experiment is to identify services with various granularities. In fact, the DFD allows the *hierarchical representation* to capture processes in different levels of abstractions (DeMarco, 1979). Therefore, by applying the proposed method to each layer of the hierarchical DFD, it is possible to extract multi-grained services.

For the experiment, we use an 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 the source code, we obtained a hierarchical DFD. Then, we applied the proposed method to the top 3 layers (Layer 0 (= Context Diagram), Layer 1 and Layer 2) of the DFD.

Figure 9 shows the services extracted from the DFD Layer 1, describing sub-systems of the whole Liquor Shop 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.

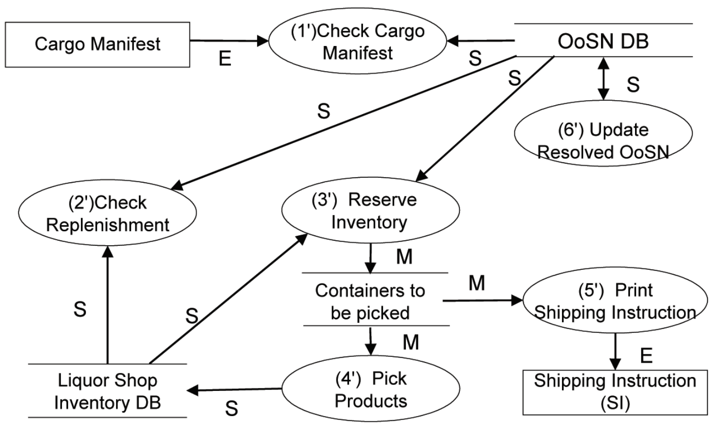
Figure 10 shows all services extracted from different layers of the DFD. In the table, the column layer represents the layer of the DFD, where 0 corresponds to the top level DFD (= context diagram), 1 corresponds to the one in Figure 9, and 2 corresponds to DFDs which refine five processes in Figure 9 (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.

EVALUATION

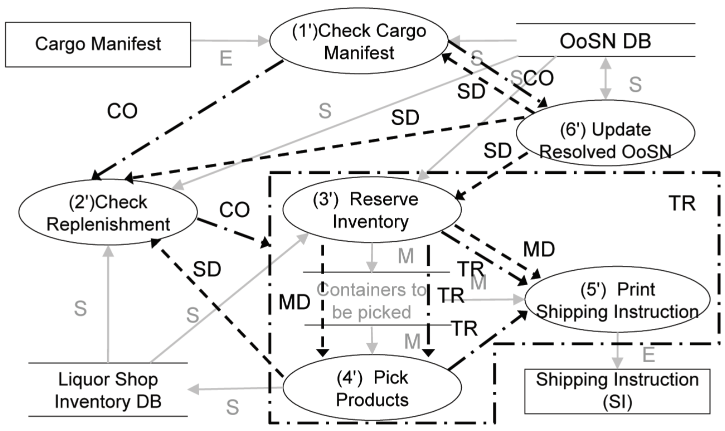
Characteristics of Extracted Services

The proposed method derives services that encapsulate control flows of internal processes as well as implementation-specific data. According to the service extraction rules, processes that require a specific execution order or transaction

Figure 7. Another implementation of ``Resolve Out of Stock" process



(a) DFD after STEP2.



(b) DFD after STEP 3.

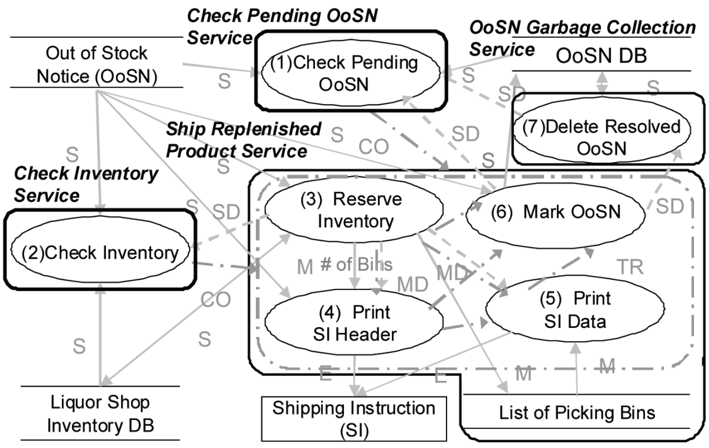
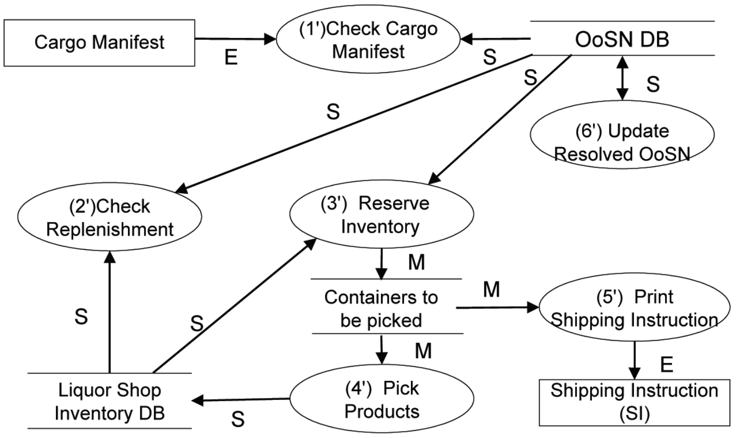


Figure 8. Service extraction from another implementation



are grouped 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* (Sneed, 2006) that simply converts the system data into an implementation-neutral form.

By the above discussion, every service obtained by the proposed method has an 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.

The proposed method can be applied to any layer of the hierarchical DFD. If it is applied to a higher layer, we can extract coarser-grained services with high-level and sophisticated functionalities. On the other hand, when it is 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 an appropriate granularity, which can satisfy Condition S3.

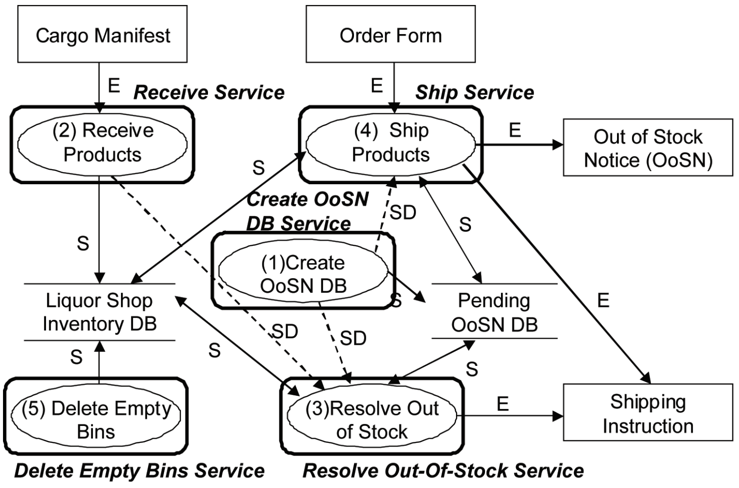
Thus the extracted services satisfy Conditions S1, S2 and S3, which can be reasonable candidates for SOA services.

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 ones 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. Note also that our method cannot cover *dynamic dependencies* which can be found only at runtime, as our dependencies are specified statically based on the source code. For such cases, 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

Figure 9. Services extraction from upper layer



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 influences 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.

Relation to Cohesion and Coupling Metrics

The principle of the service extraction rules is to aggregate multiple processes so that each group forms a *self-contained* and *loose-coupling* service. The similar principle can be found in the classical software engineering metrics: *cohesion* and *coupling metrics*.

The *cohesion* is a measure of how strongly-related and focused the various responsibilities of a software module are (Yourdon et al., 1979; Lakhotia, 1993). There are seven types of cohesion: coincidental (worst), logical, temporal, procedural, communicational, sequential and functional (best). Our interest here is to evaluate the degree of cohesion of the derived services. For each type of cohesion, we have investigated the relation to the proposed service extraction rules.

Figure 11 shows the summary. It shows an action of the proposed method for processes with a certain cohesion type. In our method, processes with coincidental cohesion have no dependency, and therefore they are separated into different services. Processes with logical or temporal cohesion may appear in the same DFD, and they are sometimes linked by execution context. In the proposed method, such processes have condition dependency (CO), and can be separated services. Processes with procedural cohesion have transaction dependency (TR), and thus must be integrated.

Processes with communicational or sequential cohesion have data dependencies. Depending on the data is system or module data, they are integrated or separated. Finally, functional cohesion appears as a single atomic process which cannot be divided further. So, we extract the process as it is. Thus, we can say that every service obtained by the proposed method has procedural or higher cohesion.

On the other hand, the *coupling* is the degree to which each program module relies on each one of the other modules (Yourdon et al., 1979; Al-Ghamdi et al., 2001). There are six

Figure 10. Multi-grained services extracted from the whole system

Layer	Service	Input	Output
0	Inventory Management	Bin Manifest Order Form	Result (true/false) Shipping Instruction Out of Stock Notice
1	Receive	Bin Manifest	Result (true/false)
2	Check Manifest	Bin Manifest	Result (true/false/failed)
2	Receive in Warehouse	Bin Manifest	Result (true/false)
1	Resolve Out of Stock	none	Result (true/false) Shipping Instruction
2	Ship Replenished Product	Out of Stock Notice	Result (true/false) Shipping Instruction
2	OoSN Garbage Collection	none	Result (true/false)
2	Check Pending OoSN	Out of Stock Notice	Result (true/false/failed)
2	Check Inventory	Out of Stock Notice	Result (true/false/failed)
1	Ship	Order Form	Result (true/false) Shipping Instruction Out of Stock Notice
2	Check Order Form	Order Form	Result (true/false/failed)
2	Check Pending Shipment	Order Form Out of Stock Notice	Result (true/false/failed)
2	Check Inventory	Order Form	Result (true/false/failed)
2	Ship Products	Order Form	Result (true/false) Shipping Instruction
2	Handle Out of Stock	Order Form	Result (true/false) Out of Stock Notice
1	Create OoSN DB	none	Result (true/false)
1	Delete Empty Bins	none	Result (true/false)

Figure 11. Relationship between cohesion metrics and service extraction rules

Cohesion Type	Dependency	Related Rules	Action
Coincidental cohesion	None	None	Separate
Logical cohesion	Control (CO)	Rule 4	Separate
Temporal cohesion	Control (CO)	Rule 4	Separate
Procedural cohesion	Control (TR)	Rule 3	Integrate
Communicational cohesion	Data (MD or SD)	Rules 1, 2	Integrate (Separate if possible)
Sequential cohesion	Data (MD or SD)	Rules 1, 2	Integrate (Separate if possible)
Functional cohesion	None	None	Extract as it is

types of coupling: content (highest), common, external, control, stamp, and data (lowest). The original definition of the coupling metric was based on the way of data passing between program modules. Hence, it has nothing to do with our definition of data dependency based on the commonality.

For instance, the external coupling occurs when two processes share an externally imposed data format (e.g., global variables), which was regarded as relatively high coupling. However, the proposed method counts the externally imposed data as the common data, which yields weak dependency (i.e., low coupling). Moreover, the data coupling, which was regarded as the lowest coupling, tends to yield strong data dependency in the proposed method, since the data is used locally by a limited number of processes. Through this investigation, we have realized that the classical coupling metric defined within the structured analysis cannot be used directly to measure the coupling between SOA services.

Related Work

Lewis et al. (2008) 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. (2007) 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 (2006) 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. (2009) 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 provides concrete rules and procedures of the service extraction, although the applications are limited to the procedural programs only.

CONCLUSION

In this paper, we have presented a pragmatic 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 granularities. Our future work includes; the refactoring method for efficient SOA migration, systematic data classification, and evaluation metrics.

ACKNOWLEDGMENT

This research was partially supported by the Japan Ministry of Education, Science, Sports, and Culture, Grant-in-Aid for Young Scientists (B) (No.217000 77).

REFERENCES

Al-Ghamdi, J., Shafique, M., Al-Nasser, S., & Al-Zubaidi, T. (2001). Measuring the coupling of procedural programs. In *Proceedings of the IEEE International Conference on Computer Systems and Applications* (pp. 297-303).

Arsanjani, A., Ghosh, S., Allam, A., Abdollah, T., Gariapathy, S., & Holley, K. (2008). SOMA: A method for developing service-oriented solutions. *IBM Systems Journal*, *47*(3), 377–396. doi:10.1147/sj.473.0377

Bell, M. (2008). *Service-oriented modeling: Analysis, design, and architecture*. New York, NY: John Wiley & Sons.

Benedusi, P., Cimitile, A., & Carlini, U. D. (1989). A reverse engineering methodology to reconstruct hierarchical data flow diagrams for software maintenance. In *Proceedings of the International Conference on Software Maintenance* (pp. 180-189).

Berkem, B. (2008). From the business motivation model (BMM) to service oriented architecture. *Journal of Object Technology*, *7*(8), 57–70. doi:10.5381/jot.2008.7.8.c6

Cetin, S., Altintas, N. I., Oguztuzun, H., Dogru, A. H., Tufekci, O., & Suloglu, S. (2007). Legacy migration to service-oriented computing with mashups. In *Proceedings of the International Conference on Software Engineering Advances* (pp. 21-21).

DeMarco, T. (1979). *Structured analysis and system specification* (pp. 409–424). Upper Saddle River, NJ: Yourdon Press.

Erl, T. (2007). *SOA principles of service design*. Upper Saddle River, NJ: Prentice Hall.

Lakhotia, A. (1993). Rule-based approach to computing module cohesion. In *Proceedings of the 15th Conference on Software Engineering* (pp. 34-44).

Lewis, G. A., Morris, E. J., Smith, D. B., & Simanta, S. (2008). *Analyzing the reuse potential of legacy components in a service-oriented architecture environment* (Tech. Rep. No. CMU/SEI-2008-TN-008). Pittsburgh, PA: Carnegie Mellon University.

Matos, C., & Heckel, R. (2009). Migrating legacy systems to service oriented architectures. *Electronic Communications of the EASST, 6*.

Newcomer, E., & Lomow, G. (2004). *Understanding SOA with web services (Independent technology guides)*. Reading, MA: Addison-Wesley.

O'Hare, A. B., & Troan, E. W. (1994) Re-analyzer: From source code to structured analysis. *IBM Systems Journals, 33*(1).

Papazoglow, M. P., & Georgakopoulos, D. (2003). Service oriented computing. *Communications of the ACM*, *46*(10), 25–28.

Sneed, H. M. (2006). Integrating legacy software into a service oriented architecture. In *Proceedings of the Conference on Software Maintenance and Reengineering* (pp. 3-14).

Yamazaki, T. (1984). Survey of program design methodologies with a common problem. *Journal of Information Processing Society of Japan*, *25*, 934–935.

Yourdon, E., & Constantine, L. (1979). *Structured design: Fundamentals of a discipline of computer program and systems design*. Upper Saddle River, NJ: Prentice Hall.

Masahide Nakamura received the BE, ME, and PhD degrees in Information and Computer Sciences from Osaka University, Japan, in 1994, 1996, 1999, respectively. From 1999 to 2000, he has been a post-doctoral fellow in SITE at University of Ottawa, Canada. He joined Cybermedia Center at Osaka University from 2000 to 2002. From 2002 to 2007, he worked for the Graduate School of Information Science at Nara Institute of Science and Technology, Japan. From 2007 to 2009, he moved to Graduate School of Engineering at Kobe University. He is currently an associate professor in Graduate School of System Informatics at Kobe University. His research interests include the service-oriented architecture, Web services, the feature interaction problem, V&V techniques and software security. He is a member of the IEEE, ACM, IEICE, and IPSJ.

Hiroshi Igaki received the BE degree (2000) in Department of Electrical and Electronics Engineering from Kobe University, Japan, and the ME degree (2002) and DE degree (2005) in Information Science from Nara Institute of Science and Technology, Japan. From 2006 to 2007, he worked for Faculty of Mathematical Sciences and Information Engineering, Nanzan University, Japan. From 2007 to 2010, he joined the Graduate School of Engineering at Kobe University. He is currently an assistant professor in Department of Computer Science, Tokyo University of Technology. His research interests include communication support in software development, home network systems and service-oriented architecture. He is a member of the IEEE, ACM, IEICE and IPSJ.

Takahiro Kimura received the ME from Nara Institute of Science and Technology, Japan in 2006. He is currently working for Nihon Unisys, Ltd., Japan. His research interests include service oriented architecture, Web service, legacy migration.

Ken-ichi Matsumoto received the BE, ME, and PhD degrees in Information and Computer sciences from Osaka University, Japan, in 1985, 1987, 1990, respectively. Dr. Matsumoto is currently a professor in the Graduate School of Information Science at Nara Institute of Science and Technology (NAIST), Japan. His research interests include software metrics and measurement framework. He is a senior member of the IEEE, and a member of the ACM, IEICE, IPSJ and JSSST.