Identifying Services from Legacy Batch Applications[*]

Raghavan Komondoor
Indian Institute of Science, Bangalore
raghavan@csa.iisc.ernet.in

V. Krishna Nandivada
Indian Institute of Technology, Madras
nvk@cse.iitm.ac.in

Saurabh Sinha
IBM Research – India
saurabhsinha@in.ibm.com

John Field
Google Inc.
jfield@gmail.com

ABSTRACT

Transaction processing is a key constituent of the IT workload of commercial enterprises (e.g., banks, insurance companies). Even today, in many large enterprises, transaction processing is done by legacy "batch" applications, which run offline and process accumulated transactions. Developers acknowledge the presence of multiple loosely coupled pieces of functionality within individual applications. Identifying such pieces of functionality (which we call "services") is desirable for the maintenance and evolution of these legacy applications. This is a hard problem, which enterprises grapple with, and one without satisfactory automated solutions.

In this paper, we propose a novel static-analysis-based solution to the problem of identifying services within transaction-processing programs. We provide a formal characterization of services in terms of control-flow and data-flow properties, which is well-suited to the idioms commonly exhibited by business applications. Our technique combines program slicing with the detection of conditional code regions to identify services in accordance with our characterization. A preliminary evaluation, based on a manual analysis of three real business programs, indicates that our approach can be effective in identifying useful services from batch applications.

Categories and Subject Descriptors

D [**2**]: 7—*Restructuring, reverse engineering, and reengineering*; D [**2**]: 13—*Reusable Software*

General Terms

Algorithms,Experimentation

1. INTRODUCTION

Legacy applications written decades ago form the backbone of the IT infrastructure of most large enterprises (e.g., as reported in a ComputerWorld magazine 2006 survey). Many of these applications are written for execution in a "batch" mode; that is, the application runs periodically according to some schedule. Whenever a business event occurs (e.g., a customer places a new order), the event generates a "transaction" (input file record), which is batched up. During application execution, all transactions that have been batched up since the previous run are processed sequentially. Interestingly, the structure of these applications—an intermingling of code fragments offering differing functionalities—reflects their usage pattern: processing multiple kinds of transactions in one go.

The presence of such loosely coupled functionalities (which can be thought of as different "services") interleaved within a single application, and even within individual programs in the application, creates challenges in comprehension, maintenance, evolution, and transformation of legacy applications [22]. In this paper, we address the problem of identifying interleaved candidate services in a batch application. The identified services enhance understandability of the application by allowing the application to be viewed as a composition of services. They can can serve as a basis for re-architecting the application to a service-oriented architecture, which is currently a largely manual, non-systematic process [15]. Finally, once migrated to a service-oriented architecture, a batch application can even be transformed into an "online" application, which is capable of processing transaction records as and when they arrive by invoking the corresponding services.

In business applications, different constituent services are typically identified by the outputs they generate. Therefore, a *naive* solution to service identification would be to apply backward slicing from each output statement in the application. However, such an approach may result in unsatisfactory results (e.g., too many overly fine-grained services). To overcome this problem, we present an approach that is based on two key insights. First, although services can be identified by the output they generate, all related output should be encapsulated within a single service. Second, different services in an application should be identified by not only the output they produce, but also by the input they consume, and by the data dependences between different services.

Illustrative Example

Consider the order-processing application shown in Figure 1, which we use as the running example in this paper. It is written in a simplified variant of the Cobol language, which is the predominant language of legacy business applications. The initial part of the program (`DATA DIVISION`) contains declarations of files and variables. Variables are prefixed by *level numbers* (e.g., `01` or `05`), which serve to indicate nesting, akin to record-field relationships, among variables. Note that Cobol does not require statements to refer to a fully qualified data name when an unqualified name is unambiguous. The subsequent part of the program contains the

[*]Part of this work was done when the first, second, and fourth authors were affiliated with IBM Research

```
DATA DIVISION.
FILE SECTION.
FD in-file ACCESS IS SEQUENTIAL.
  01 in-rec PIC X(15).
FD out-file ACCESS IS SEQUENTIAL.
FD item-table ACCESS IS RANDOM.
  01 item-rec.
    05 item-id PIC 9(4) IS PRIMARY KEY.
    05 item-avbl-count 9(6).
WORKING-STORAGE SECTION.
01 header-rec.
  05 header-status PIC x(8).
  05 FILLER PIC X(7).
01 order-rec.
  05 ord-type PIC X(7).
  05 ord-id PIC X(6).
  05 ord-num-items PIC 9(2).
01 i PIC 9(2).
01 ord-item-rec.
  05 ord-it-id PIC 9(4).
  05 ord-it-count 9(4).
  05 FILLER PIC X(7).
01 full-all-avbl PIC X(5).
01 ord-item-recs OCCURS 100 TIMES.
  05 ord-its-id PIC 9(4).
  05 ord-its-count 9(4).
  05 FILLER PIC X(7).
01 num-fulfilled PIC 9(15).

PROCEDURE DIVISION.
/ 1/ OPEN INPUT in-file item-table
        OUTPUT out-file.
/ 2/ READ in-file.
/ 3/ MOVE in-rec TO header-rec.
/ 4/ MOVE 'received' TO header-status.
/ 5/ WRITE header-rec TO out-file.
/ 6/ MOVE 0 TO num-fulfilled.
/ 7/ WHILE in-file NOT AT END
/ 8/   READ in-file INTO order-rec
/ 9/   IF ord-type = 'regord'
/10/     ADD 1 to num-fulfilled
/11/     WRITE order-rec TO out-file                           (3)
/12/     i = 1                                                 (3)
/13/     WHILE i <= ord-num-items                              (3)
/14/       READ in-file INTO ord-item-rec                    (1,3)
/15/       LOOKUP item-table KEY IS ord-it-id INTO item-rec  (1,3)
/16/       IF item-avbl-count < ord-it-count                 (1,3)
/17/         MOVE item-avbl-count TO ord-it-count.           (1,3)
/18/       WRITE ord-item-rec TO out-file                    (1,3)
/19/       i = i + 1                                           (3)
/20/     END-WHILE                                             (3)
/21/   ELSE IF ord-type = 'fullord'
/22/     MOVE 'ok' TO full-all-avbl                            (2)
/23/     i = 1                                                 (2)
/24/     WHILE i <= ord-num-items                              (2)
/25/       READ in-file INTO ord-item-recs(i)                  (2)
/26/       LOOKUP item-table KEY IS ord-its-id INTO item-rec   (2)
/27/       IF item-avbl-count < ord-its-count(i)               (2)
/28/         MOVE 'notOk' TO full-all-avbl                     (2)
/29/       ENDIF                                               (2)
/30/       i = i + 1                                           (2)
/31/     END-WHILE                                             (2)
/32/     IF full-all-avbl = 'ok'                               (2)
/33/       ADD 1 to num-fulfilled
/34/       WRITE order-rec to out-file                         (2)
/35/       i = 1                                               (2)
/36/       WHILE i <= ord-num-items                            (2)
/37/         WRITE ord-item-recs (i) TO out-file               (2)
/38/         i = i + 1                                         (2)
/39/       END-WHILE                                           (2)
/40/     ENDIF                                                 (2)
/41/   ENDIF
/42/ END-WHILE
/43/ WRITE 'Num. orders fulfilled = ', num-fulfilled TO out-file.
```

Figure 1: Example business application.

PROCEDURE DIVISION.[1]

The program has a *main* loop (lines 7–42). In each loop iteration, the program processes a transaction (an order) from the input file. A transaction consists of an order "header" record (read into variable order-rec in line 8), followed by a number of item records, one for each item that is included in the order (the number of items in the order is available in the field ord-num-items of order-rec). There are two types of orders: "regular" orders, handled in lines 10 through 20 (then branch of the if conditional in line 9), and "full" orders, handled in lines 22 through 40 (else branch). In both alternatives, an inner loop processes the multiple items that are part of an order. After processing an order, the program writes it out to the file out-file (lines 11,18, 34 and 37).

This program is a typical example of a business application, which typically has a loop that reads transaction records from an input file, and "dispatches" each transaction to an appropriate region of code within the loop body depending on the type of transaction. The code region applies some business logic on the transaction, perhaps using an inner loop to process the constituents of the transaction, creates one or more output records, and writes them out. Intuitively, different collections of these regions correspond to different services that one may wish to identify.

There are different candidate services in the example program in Figure 1 which the developer may want to explore. For instance, consider two alternative sets of services: (i) a service for processing a full order and a service for processing a regular order; and (ii) a service for processing a full order, a service for processing an item in a regular order, a service for processing a regular order, and a service for processing all orders (i.e., the entire program). The second alternative captures the hierarchical nature of the underlying business logic and, therefore, the services. Each service has an *output seed* (one or more WRITE statements) and may have one or more *sources* (input buffers of READ statements whose values are used by the service).

We illustrate *some* of the services identified by our approach on the example in Figure 1 by labeling all statements belonging to a service with an identifying label on the right-hand side. For example, the statements labeled (2) belong to a service.

Code labeled (1). These statements constitute a service that processes an item from a regular order. For this service, the output seed is statement 18, whereas the source is statement 14.

Code labeled (3). Using statements 11 and 18 as the output seed and lines 8 and 14 as the input source, the entire logic for processing a regular order can be identified as one service. In other words, the approach enumerates two services seeded at line 18, labeled (1) and (3), with (1) being contained within (3).

Code labeled (2). Using lines 34 and 37 as the output seed and lines 8 and 25 as the input source, the approach identifies lines 22–40 (except the update to num-fulfilled in line 33) as a candidate service, to process a "full" order. This service corresponds to the business logic that, in a full order, all items must be processed together (by taking into consideration whether all the requested items are available in requisite quantities). Note that, although both the inner loops are included in the candidate service, line 33—which lies between the two loops, but is not pertinent to the logic for processing an individual order (it increments the count of fulfilled orders)—is excluded from the service.

Our approach also identifies services that contain output statements 5 and 43. These are "header" and "trailer" output statements, not present in any loop in the batch program. These output state-

[1]For readability, we use the intuitively simpler syntax of WHILE loops instead of the PERFORM UNTIL loops of Cobol.

ments are purely artifacts of the batch program and, therefore, are not particularly interesting as separate services. We indicate in Table 1 (Section 3.3) the set of *all* services identified by our approach in this example. Note that the services we identify are basically *candidate* services. It is up to the develop to choose a set of services from among the ones emitted for further analysis and use.

Our Approach

Our goal is to identify candidate code fragments for extraction into services (as illustrated using the example in Figure 1). To develop a systematic approach for service identification, we provide a formal characterization of candidate services in terms of input/output characteristics, control-flow properties, and data-flow properties. The characterization is tailored to the coding idioms commonly present in transaction-processing business applications.

One of the key novel aspects of the characterization is that it is based on the notion of a *tag conditional*, which is useful for separating independent services. A tag conditional is an `if` or `switch` statement that checks the *tag field* of a record to determine the record type. For example, `ord-type` is a tag field in the example program, and the conditionals in lines 9 and 21 check this field to determine the type of an order. Intuitively, a tag conditional controls a region of code that is likely to contain an independent service. For instance, line 9 controls the region that contains the "regular" order service, whereas line 21 controls the region that contains the "full" order service.

We present a static-analysis approach that identifies candidate services according to our characterization. The approach analyzes each code region r that is controlled by a tag conditional or a loop header. It first identifies the output seed (i.e., set of `WRITE` statements) in r; then, it considers different bounding regions that surround r and, for each bounding region b, obtains a *backward slice* from the seed restricted to b. It emits the slice as a service, provided the incoming values to and outgoing values from the slice satisfy certain properties (this is to minimize coupling between the service and its context). By exploring different bounding regions for each same seed region r, the analysis is able to generate services at different levels of granularity, and let the programmer finally pick the ones that they consider the most appropriate.

To evaluate the effectiveness of our approach, we performed a preliminary evaluation using three real-world business programs written in Cobol. In the study, we manually applied our approach to the three programs to identify services, and compared our approach with the naive approach of performing slicing from individual `WRITE` statements. For each subject, our approach identified services more accurately—i.e., services that match closely the underlying intuitive service structure—than the naive approach. Our study also illustrates that the naive approach can not only identify overly fine-grained services, but also miss identifying useful services.

The main benefit of our approach is that it provides a systematic and structured way of identifying services. Moreover, it identifies services that have several desirable properties, such as cohesiveness of a service, independence (or loose coupling) among different services, and a hierarchical service structure.

The key contributions of the paper are

- A formal characterization that can be used to identify candidate services from transaction-processing business applications
- The presentation of a novel static analysis, based on tag-condition analysis and program slicing, for identifying candidate services.
- The description of three applications of service identification that are important from the application-developer perspective: (1) program understanding, (2) modularization of business applications, (3) batch-to-online translation
- The presentation of an empirical study of the effectiveness of our technique using three real Cobol batch programs.

In this paper, we target transaction-processing business applications, and leave as future work the study of service identification for other domains, such as web applications.

The rest of the paper is organized as follows. We present our assumptions and background definitions in Section 2. Following that, we present our characterization of services (Section 3), describe the algorithm for identifying services (Section 3.3), and discuss three applications of services identification (Section 5). In Section 6, we present the empirical study. Finally, we conclude with a discussion of related work in Section 7, and directions for future work in Section 8.

2. ASSUMPTIONS AND DEFINITIONS

We assume a simple imperative language, with the usual constructs, such as assignments, sequencing, conditional statements, loops, jumps (i.e., `goto` statements), `READ` and `WRITE` statements to access sequential files, `LOOKUP` and `UPDATE` statements to access random-access files, and procedure calls. Variable declarations (as in Cobol) are of the form "*var* `PIC X(`n`)`" or "*var* `PIC 9(`n`)`," which declare the variable *var* to store a byte sequence of length n. An `X` indicates that the bytes store arbitrary characters, whereas a `9` indicates that the bytes store decimal digits. Conditional statements are assumed to be side-effect free; if a conditional has side effects, it ought to be first transformed to remove the portions with side effects into separate statements. We assume that the Cobol `PERFORM` loops have been translated to normal `while` loops, with the introduction of explicit statements, if necessary, to initialize and increment loop-induction variables.

Input (i.e., `READ`) and output (i.e., `WRITE`) statements have semantics as described below. Each statement refers to a buffer (i.e., variable) and a file. When the statement executes, bytes are transferred between the buffer and the file. The number of bytes transferred is equal to the declared length of the buffer. In the example program of Figure 1: (a) Line 8 reads the next data record from the sequential input file `in-file` into the input buffer (variable) `order-rec`; (b) Line 15 reads the record that has value `ord-it-id` in its primary key field from the random-access database table `item-table` into the variable `item-rec`; and (c) Line 11 appends the value (scalar or record) in `order-rec` to the sequential output file `out-file`. For ease of presentation we assume that distinct `READ` statements use distinct input buffers, and that other statements do not modify input buffers. We use the terms "record" and "value" interchangeably because, at a basic level in Cobol, all values are simply sequences of bytes, with no real difference in representation between a scalar value and a record value.

In our approach, a program is represented as a *control-flow graph*, in which nodes represent statements or conditionals, and edges represent potential flow of control among the statements/conditionals.

Our approach requires the computation of conservative interprocedural data dependences [16]. A statement u is *data dependent* on statement d if d assigns a value to a memory location l, u reads l, and there exists a path from d to u along which no statement assigns a value to l. A statement s is *control dependent* on a conditional c if and only if there exist multiple branches from c such that along one of the branches s is definitely reached, whereas along all other branches s may not be reached [8]. A statement *s dominates* statement t if and only if all paths from the entry of the program to t go through s [1]. A *backward slice* of a program from a node s is the set of nodes on which s is dependent by the transitive closure of data and control dependences [20]. To account for jump nodes correctly during slicing, we treat jump nodes as *pseudo predicates* to ensure that they are included in a slice as appropriate [3].

An *e-hammock* [13] is a subgraph of a CFG that has a single entry node, and, if all jumps are replaced by no-ops, a single fall-through exit node outside the subgraph. It can be shown that a CFG subgraph is an e-hammock if and only if the subgraph corresponds to a sequence of source-code statements (simple or compound) having the additional property that its entry node is the target of all incoming jump edges (i.e., those whose sources are outside the block sequence). An e-hammock H is the *tightest e-hammock* surrounding a set of CFG nodes N if and only if H includes all the nodes in N, and $\forall H'$ such that H' is an e-hammock and H' includes all the nodes in N, $H' \supseteq H$.

We handle only programs without improperly overlapping loops (i.e., *reducible* programs). Although arbitrary programs involving `goto` statements can be irreducible in general, these are very rare in practice.

3. CHARACTERIZATION OF SERVICES

3.1 Informal Characterization of a Service

A service S in a program P is a fragment of code (i.e., a subset of CFG nodes) that satisfies certain properties. The properties are of two kinds: *control-flow properties* and *data-flow properties*. Our intuition is that fragments that satisfy these properties tend to provide meaningful, independent functionality. Here we provide informal introductions to these properties; we formalize these notions in Section 3.3.

The nodes in a service need not necessarily constitute a contiguous, single-entry single-exit region of the CFG. The required control-flow properties, intuitively, are (1) the service be *contained* in an e-hammock in the CFG (which is basically a CFG subgraph that corresponds to a sequence of source-level statements with no incoming jumps into the middle of the sequence), (2) the entry node of this region be included in the service (thus becoming the entry node of the service), and (3) each conditional node in the region that *controls* one more nodes belonging to the service also be in the service. The control-flow properties basically constitute a part of the sufficient condition for extractability of the service into a separate procedure [13].

Each time controls enters the entry node of the service S during the execution of P we call it an *invocation* of the service. Note that there may be intervening nodes in the region containing a service that do not belong to the service; for instance, the statement labeled 33 surrounded by service (2) in Figure 1. These may cause control to leave a service and then re-enter it (via non-entry nodes) within a single invocation of the service; e.g., in the example control may leave service (2) at line 32 and re-enter it at statement 34. This is purely an artifact of interleaving in the given source code, and will vanish if the interleaved services are separated by a source-to-source transformation (see further discussion in Section 5).

The data-flow properties ensure that the service interacts with its context (i.e., surrounding code) in restricted ways, thus minimizing coupling with the environment. The properties are with respect to the *source* variables as well as *output* variables of the service. A source variable of a service is a variable through which a value generated outside the service (or in a previous invocation of the same service) flows into a computation in the current invocation of the service. Intuitively, the sources of a service are its parameters from its context. We require that each source variable v is (1) an input buffer of a `READ` statement, (2) a variable that is declared as "global" by the programmer, meant for communication between services or between a service and its context, or (3) derives its value from the current values of other variables in categories (1) and (2). These restrictions ensure the *independence* of each service invocation from its context (i.e., the surrounding code and other service invocations). By restricting a service to use (1) the values in input buffers, which are basically inputs to the entire program from its environment, and were not computed by the context of the service, (2) the values in global variables designated for inter-service communication, and (3) variables whose values are derived from these (and *not* other arbitrary variables whose values are defined in the context of the service), we seek to minimize coupling between a service and its context. Note that in addition to receiving values through its sources, a service may also contain `READ` statements through which it directly reads values from input files.

Furthermore, in order to satisfy the normal meaning of a parameter, we require that each source of a service must either remain invariant during the execution of the service, or must be written to only by statements within the service. That is, any statement that is not in the service but intervenes between statements in the service should not update any source. We call this the "unique parameter value" requirement. In the interest of space, we postpone discussion about *output variables* to Section 3.3.

Consider the three services marked in Figure 1. Assume that the programmer has specified `item-table` (the persistent table that contains information on available items) as a global variable. In this example, none of the services have any output variables. For services (2) and (3), the source variables are the input buffer in line 8, whose fields are used in both services, as well as `item-table`. For service (1) the only source is `item-table`. Each of these services also reads order item records from the input file directly. Note that services (2) and (3) read multiple order items in each invocation; although we might want to consider having the context read these items from the input file and pass them as parameters to the services, this would require an invasive source transformation.

```
/ 1/ balance = 0
/ 2/ WHILE in-file not AT END
/ 3/   READ in-file INTO trans
/ 4/   IF trans.code = 'depo'
/ 5/     balance = balance + trans.amt
/ 6/     WRITE 'deposit success', balance
/ 7/   ELSE IF trans.code = 'withdr'
/ 8/     IF balance >= trans.amt
/ 9/       balance = balance - trans.amt
/10/       WRITE 'withdraw success', balance
/11/     ELSE
/12/       WRITE 'withdraw reject', balance
/13/     ENDIF
/14/   ENDIF
/15/ END-WHILE
/16/ WRITE balance
```

Figure 2: Example to illustrate the specification of global variables. (Data declarations are omitted for brevity.)

3.2 Specification of Global Variables

Persistent tables, such as `item-table` in the example in Figure 1, normally ought to be declared as global variables. As regards other program variables, we illustrate the need for programmers to specify explicitly the ones that are to be treated as global variables. Consider the example in Figure 2. This program updates the bank balance of a customer by processing the transactions from `in-file`. Lines 5–6 and lines 8–13 correspond to the candidate services that process an individual deposit and withdrawal transaction, respectively. Lines 4–14 correspond to the candidate service "process individual transaction," whereas the entire program corresponds to the candidate service "process all transactions."

The global-variables specification captures the programmer's intent of how independent the inferred services ought to be and, indirectly, the granularity of the services to be inferred. For the example in Figure 2, consider the three finer-grained candidate services that are contained within the loop body (i.e., the services other than "process all transactions"). Suppose the programmer does not declare `balance` to be a global variable. Then, variable `balance`, which is a source, satisfies neither of the three properties of sources mentioned in Section 3.1. In particular, it does not satisfy the third property because its value is dependent on the *old* values in the input buffer `trans` that were read in iterations prior to the current one. Therefore, without the declaration of `balance` as a global variable (which would cause it so satisfy the second property), the three finer-grained services would be not identified.

3.3 Formal Definition of a Service

Before presenting our definition of a service, we present preliminary definitions. In the following, we use S to refer to the code fragment (i.e., a set of CFG nodes) in a candidate service.

DEFINITION 1. **(Downwards-exposed variable)** *A variable v is said to be* downwards exposed *in S, or equivalently an output variable of S, if some definition of v in S reaches a use of v outside S, or reaches a use of v in S via a path that leaves S and then re-enters S through its* entry *node (i.e., the entry node of the tightest e-hammock that contains S).*

DEFINITION 2. **(Upwards-exposed variable)** *A variable v is said to be* upwards exposed *in S, or equivalently a source variable of S, if some use of v in S is reached by a definition of v outside S, or is reached by a definition of v in S via a path that leaves S and then re-enters S through its entry node.*

DEFINITION 3. **(Incoming value)** *A value in variable v is said to be* incoming *to S if the value was placed in v either by a statement not in S or by a node in S during a previous invocation of S, and resides in v at some point of time t in execution when control enters a node in S from a node outside S. If another variable w also contains an incoming value at time t, the two incoming values are said to* correspond.

Note that the statement that places the value in v need not precede the entry of S; it could be an intervening statement, in which case the node to which control enters at time t would not be the entry node of S.

DEFINITION 4. **(Derived variable)** *A variable v is said to be* derived *from a set of variables V, if whenever there is an incoming value in v to S, there are corresponding incoming values in all variables in V to S, and the incoming value of v is fully determined by (i.e., is a mathematical function of) the corresponding incoming values of the variables in V.*

We assume that each variable in the program (including input buffers) have a special value *Uninit* before the program starts execution.

DEFINITION 5. **(Service)** *A* service *S in a program P is a tuple (S, G, e), where S is a (possibly non-contiguous) set of nodes, G is the set of (programmer-declared) global variables, excluding input buffers, in P, and e is the entry node of S such that the following control-flow and data-flow properties are satisfied.*

- (Control-flow property 1) *Let H be the* tightest *e-hammock that contains the nodes in S. Then, for each $n \in S$, all control-dependence ancestors of n in H are also in S. The entry node of H is the entry node of the service.*
- (Data-flow property 1) *Each upwards (downwards) exposed variable v (1) belongs in G, or (2) is an input buffer whose `READ` statement in outside (inside) S, or (3) is derived (Definition 4) from a subset of input buffers and global variables that are upwards (downwards) exposed in S.*
- (Data-flow property 2) *For each upwards-exposed variable v there does not exist any execution of P wherein during a single invocation of S two different incoming values of v are used by nodes in S during this invocation of S.*

Note that data-flow property 2 formalizes the "unique parameter value" requirement discussed in Section 3.1.

```
Input. program $P$, declared set of global variables $G$.
(Step 1) Identify all tag fields and tag conditionals in $P$.
(Step 2) Identify the set of regions $Reg$ in $P$.
(Step 3) For each region $reg \in Reg$:
    (a) Let $W$ be the set of all WRITE statements in $reg$ Proceed to
        Step (b) if $W$ is non-empty.
    (b) For each region $reg_a$ that is equal to or contains $reg$
        (i) Obtain a backward slice $S$, bounded within region $reg_a$,
            by using the nodes in $W$ as the slicing criterion. A re-
            striction observed during the slicing is that we do not
            follow dependence edges backward that are induced by
            paths not entirely contained within $reg_a$.
        (ii) Emit $(S, P, G)$ as a service provided
            A. No other service has been emitted so far that differs
               from this one only in terms of have more or fewer
               "if" conditionals, and
            B. It satisfies the characterization in Section 3.3.
```

Figure 3: The algorithm for identifying services.

4. DESCRIPTION OF THE ALGORITHM

Figure 3 presents the algorithm for identifying services that conform to the characterization presented in the previous section. The algorithm takes as inputs the program P and the set of global variables G. It generates as output a set of candidate services. Next, we discuss different steps of the algorithm.

Step 1: Identify tag fields and conditionals

Tag conditionals are the "if" conditionals that test a *tag field* in an input buffer, or a variable that stores a copy of a tag field of an input buffer, against one of the possible tag values of that field. A tag field is basically a field in an input record that determines how the record ought to be processed (e.g., field `ord-type` in the example in Figure 1). The algorithm first identifies *tag fields* using the heuristic presented in Reference [14]; alternatively, tag fields could be specified manually using programmer annotations. From the tag fields, the algorithm determines the variable occurrences—a variable occurrence is a pair (n, v), where n is a node and v is a variable—to which tag fields of input buffers may flow via zero or more copy statements. To do this, the algorithm uses the transitive data-dependence analysis described in Reference [14]. Finally, the algorithm characterizes as a tag conditional any "if" conditional p that contains a conjunct of the form $v = k$ or $v \neq k$, such that (p, v) was previously identified as storing a copy of a tag field. Although this technique may not necessarily yield all conditionals in a program that a human would regard as a tag conditional, for our analyzed subjects in the empirical study (Section 6), it identified all tag conditionals.

EXAMPLE 1. Consider the example program in Figure 1, in which the algorithm identifies `order-rec.ord-type` as a tag field. Using this information, the algorithm identifies the conditionals in lines 9 and 21 as tag conditionals. □

Step 2: Identify regions in the program

This step decomposes the program P into a tree structure of *regions*; each region is a maximal e-hammock that is controlled by a branch of a tag conditional or a loop header (the tag conditional or the loop header is not part of the region). Note that the outermost region is the entire program and has no parent region.

EXAMPLE 2. For the example in Figure 1, the algorithm identifies seven regions: (R_1) the whole program, (R_2) the loop body in lines 8–41, (R_3) lines 10–20, (R_4) lines 14–19, (R_5) lines 22–40, (R_6) lines 25–30, and (R_7) lines 37–38. The containment hierarchy among these regions is $R_1 \rightarrow R_2$, $R_2 \rightarrow R_3$, $R_3 \rightarrow R_4$, $R_2 \rightarrow R_5$, $R_5 \rightarrow R_6$, and $R_5 \rightarrow R_7$. □

Step 3: Perform slicing in each region

Step 3 performs backward slicing in each region to identify candidate services. For each region *reg*, this step first identifies the set W of `WRITE` statements in *reg*. Next, it computes a backward slice using the statements in W as the slicing criteria. To do this, the algorithm identifies a bounding region reg_a of *reg*, which may be the same as *reg*. By varying the bounding region, multiple candidate services are identified (from which the programmer could select the desired ones). We illustrate the benefit of this later in this section. In addition, we use a heuristic to prune away certain candidate services to reduce seemingly similar services: if a candidate service from an outer boundary differs from a candidate service from an inner boundary only in containing some additional "if" conditional nodes (i.e., no additional computation nodes), we ignore the outer candidate service and consider only the inner one.

EXAMPLE 3. As mentioned earlier, `order-rec.ord-type` is the only tag field in the program in Figure 1, and `item-table` is the only variable declared as global by the user. Suppose that *reg* consists of lines 14–19. Then, $W = \{18\}$. We illustrate different services that can be identified by considering different bounding regions. First, let the bounding region reg_a be the same as *reg*. The algorithm computes a candidate service that is this entire region (see the fragment labeled (1) in Figure 1). This corresponds to the service for processing an item of a regular order. Next, let reg_a be the containing region 10–20. In this case, the algorithm computes the service consisting of lines 12–20. This candidate is a variant of the ideal service that processes an entire regular order (the ideal one from the programmer's perspective would be the service labeled (3) in Figure 1, which includes the `WRITE` statement in line 11). These two candidate services differ not in the set of `WRITE` statements they contain, but in that the second candidate includes a loop (lines 13–20) that is not part of the first candidate. This is the motivation for us to generate multiple candidate services starting from the same seed of `WRITE` statements W: we would like to report services that process records at different levels of granularity, and let the programmer decide which one (or many) among them are the most desirable. Next, by setting the bounding region reg_a as lines 8–41, we get a candidate containing lines 8–9 and lines 12–20. This is yet another variant of service (3) in Figure 1. Finally, by setting reg_a to be the entire program, we get a candidate consisting of the lines 1, 2, 7–9, 12–20, and 42 (the end of the outer loop). This essentially corresponds to a partial evaluation of the entire program, restricted to processing all regular orders from an input file that contains both regular orders and full orders. □

No.	Seed Region (*reg*)	Bounding Region (reg_a)	Code in Service	Description
1	14–19	14–19	14–18	reg. order item
2	14–19	10–20	12–20	reg. order
3x	14–19	8–41	8, 9, 12–20	reg. order
4	14–19	P	1, 2, 7–9, 12–20, 42	all reg. orders
5	10–20	10–20	11–20	reg. order
6x	10–20	8–41	8, 9, 11–20	reg. order
7	10–20	P	1, 2, 7–9, 11–20, 42	all reg. orders
8	37, 38	37, 38	37	—
9	37, 38	22–40	22–32, 35–40	full order
10x	37, 38	8–41	8, 21–32, 35–40	full order
11	37, 38	P	1, 2, 7–8, 42, 21–32, 35–40	all full orders
12	22–40	22–40	22–32, 34–40	full order
13x	22–40	8–41	8, 21–32, 34–40	full order
14	22–40	P	1, 2, 7–8, 42, 21–32, 34–40	all full orders
15	8–41	8–41	8, 9, 11–32, 34–41	reg./full order
16	8–41	P	1, 2, 7–8, 42, 9, 11-32, 34–41	all orders
17	P	P	P	all orders

Table 1: Candidate services for the example in Figure 1.

Table 1 lists all candidate services for the example program identified by our approach; the four services discussed in Example 3 are shown in the first four rows of the table. The first column of the table is the serial number (we explain later the meaning of annotation 'x'). Columns 2 and 3 show the seed region and the bounding region, respectively. Column 4 shows the code in the identified service. Finally, Column 5 provides an informal description of the closest ideal service. The rows in the table are sorted first by the seed region (column 2), and within each seed, by increasing sizes of the bounding regions (column 3).

The first, fifth, and the twelfth services are the ones marked in Figure 1 as services (1), (3), and (2), respectively. Note that one of the candidate services computed by the algorithm, namely service 8, has no meaningful description and is, therefore, an undesirable service. Rows 1, 5, 7, 12, 14, 15, and 17 show the ideal services from the programmer's perspective, whereas the other services (except the one in row 8) are variants of these ideal services.

The candidate services in rows 3, 6, 10, and 13 (annotated with 'x') are eliminated by the heuristic that checks whether these services differ from other candidates only in terms of having extra conditionals. For instance, the candidate in row 3 is eliminated because it is identical to the one in row 2 except that it has an extra conditional node, namely, line 9. Thus, four of the 17 candidate services identified in Step 3(b)(i) are eliminated by this heuristic.

Next, the algorithm checks if each candidate service satisfies the characterization of Section 3.3. All candidates identified by the algorithm satisfy the control-flow property. In general, they may not satisfy the data-flow properties. However, all 17 candidate services identified by the algorithm on the running example satisfy the characterization. In fact, as we observe in Section 6, in practice, the algorithm almost always identifies only candidate services that meet the characterization, *provided* the programmer declares an appropriate set of variables as global variables. For instance, as discussed in Section 3.2, if `balance` were to be not declared a global variable, only the entire program shown in Figure 2 satisfies the characterization. However, if `balance` is declared as a global variable, various meaningful fragments within the program (e.g., lines 8–13) satisfy the characterization.

In other words, the characterization is useful in two ways: (1) to check whether any given arbitrary fragment of code (not necessarily identified by the algorithm) is a service, and (2) as a feedback mechanism to the programmer during the execution of the algorithm, to fine tune the set of variables declared as global, until the candidate services that they would like meet the characterization.

Discussion

It follows from the previous discussion that the number of services identified by the algorithm can be up to $|reg|^2$, where $|reg|$ is the total number of regions in the program. The total number of regions itself is bounded by the number of loops and tag conditionals in the program.

The services emitted by our algorithm form a hierarchical tree structure. Basically, a service S_1 is contained in another service S_2 if the set of nodes in S_1 is contained in the set of nodes in S_2. For example, service (1) in Figure 1 is contained within service (3).

Note that any single service we identify is contained within a single procedure; this service may, however, contain procedure calls. We leave it to future work to extend the approach to identify inter-procedural services.

5. APPLICATIONS OF SERVICE IDENTIFICATION

In this section, we discuss three applications of our service identification technique: program understanding, batch modularization, and batch-to-online transformation.

5.1 Program Understanding

It is well-recognized that identifying and labeling *concepts* in an application improves its understandability [21]. The services we identify within batch applications are an instance of concepts. The hierarchy of services our algorithm emits can be represented as a service tree within an integrated development environment (IDE), such as Eclipse, which can be enabled as part of a new *service view*. This service view can color code (and indent) individual services so as to help the developer focus on a service, and edit just that particular service. If the update to a particular service can change the service tree, it gets reflected in the tree and keeps the developer aware of any changes that might reorganize the services. Additionally, the developer can annotate different services with comments and the IDE can link the annotations to the service nodes.

5.2 Batch Modularization

An interesting application of our service-identification technique is that of re-engineering legacy code by extracting different services as separate procedures, and replacing the previously in-place code of each service with a call to the extracted procedure. Although this does not modify the essential batch nature of the application, it would improve the understandability and maintainability of a batch program. The application developer may select a service from the ones identified in Section 3.3, and pass it onto an existing procedure-extraction algorithm (e.g., [13, 17]). The procedure extractor could rewrite the batch program by creating a separate procedure corresponding to each service. Because of the hierarchical nature of the services, an extracted procedure may contain calls to other previously extracted services. The extraction algorithms are quite general; for instance, for the example in Figure 1, the algorithms can extract contiguous fragments, e.g.,service (3), as well as non-contiguous fragments, e.g.,service (2). Service (2) would be extracted by first placing a copy of line 32 after line 40, then moving line 33 so that it becomes nested under this new copy, then extracting the (now contiguous) fragment. An interesting challenge, which we leave to future work to address, is that of automatic naming of the services.

5.3 Batch-to-Online Transformation

Advances in the computing industry and business requirements have in many situations made it critical to translate legacy batch systems into online systems, in which individual services can run independently of each other and process transactions as and when they arrive. Service identification is the first step toward such a batch-to-online translation. Once services have been identified, each one can be extracted and wrapped as a separate execution unit. These execution units can be called asynchronously as the inputs to them become available from the environment. They can be called concurrently too, provided the critical sections that access shared data are protected by appropriate synchronization.

The hierarchical organization of our services enables the online system to have an interesting property. An "outer" service can be invoked whenever its "header" transaction arrives, and then suspended pending the arrival of the "inner" transactions that are conceptually nested within it. Whenever an inner transaction arrives, the corresponding inner service can be invoked in the context of the suspended invocation of the outer service. Finally, after all inner transactions have arrived and have been processed, the outer service can be resumed and allowed to complete. For instance, in the example in Figure 1, the environment can invoke service (3) whenever the header of an order record arrives. It would then suspend this invocation (after line 12), and invoke service (1) (in the context of the suspended service (3)) whenever a transaction corresponding to an item within the order arrives. Finally, it would complete the suspended invocation of the outer service (i.e., service (1)) after all items have arrived. With this approach the system will be able to process multiple orders concurrently, even if the items within each order arrive non-deterministically from the environment, and interleaved with items of other orders. We leave it to future work to flesh out the details of this particular application, and validate it in real contexts.

6. EVALUATION

We have presented a novel static-analysis-based approach for identifying services from batch-processing business applications. To evaluate the approach, we performed a manual analysis of the three medium-sized proprietary Cobol programs, each around 1000 lines long. These are typical examples of batch programs that we have encountered in many enterprise applications. Each of these batch programs has a single outermost loop, which processes all input records. Each program consists of a sequence of "Cobol paragraphs", which are "performed" (invoked) at different places. For the purpose of this evaluation, we assumed that all the paragraphs were inlined. For each benchmark, we present a few statistics: number of lines of Cobol code, number of `WRITE` statements, number of loops, number of marked global variables, and the number of services identified by our approach. For each benchmark, we contrast our approach with the naive approach of performing backward slicing using each `WRITE` statement as a seed. We discuss our experience with each of the program, followed by some general observations.

Benchmark A1

Lines = 1323; *WRITEs* = 62; *Loops* = 5; *Marked global vari-*

```
/ 1/ Initialization();
/ 2/ while (...) {
/ 3/        Read-and-validate-input-record();
/ 4/        switch(inp-rec.f1) {
/ 5/        // 12 way switch statement
/ 6/            case 1: ...
/ 7/            case 2: ...
/ 8/            ...
/ 9/            case 7:
/10/                    switch(inp-rec.f2) {
/11/                        case 'a': ...
/12/                        case 'b': ... }
/13/            ...
/14/            case 12:
/15/                    switch(inp-rec.f2) {
/16/                        case 'a': ...
/17/                        case 'b': ... }
/18/        }
/19/ }
/20/ Write-statistics();
```

Figure 4: Skeleton code of Benchmark A1.

Service	Seed Region	Bounding Region	Code in Service	Description
1	6	3–18	3,4,6,18	*ReadValidateReformat1*
2	6	1–20	1–4,6,18,19	*Reformat1_InLoop*
…	…	…	…	…
35	17	3–18	17	*ReadValidateReformat18*
36	17	1–20	1–4,17–19	*Reformat18_InLoop*
…	…	…	…	…
37	3–18	3–18	3–18	*ReformatAnyOneRec*
38	3–18	1–19	1–19	*ReformatAllRec*
39	1–20	1–20	1–20	*A1*

Table 2: Candidate services for Benchmark A1.

ables = 0; *Identified services* = 39.

A skeleton of Benchmark A1 is shown in Figure 4. This program is a banking information system that formats input files and writes to an output file based on different input parameters (e.g., new applicant, deposit, etc.). There are 12 types of records; each type is processed in a case statement of a 12-way `switch` statement. Some of these 12 record types (case 7 through case 12) have further variations based on other input-record fields. Of these total 24 switch branches, there are 18 *logical* switch branches in the program, including the outer switch with 6 branches (case 1 through case 6), and 12 inner ones within some of the outer-level branches (for instance, `inp-rec.f1` is `7` *AND* `inp-rec.f2` is `'a'`). Of the total 62 `WRITE` statements, 8 occur in `Write-statistics` module, 36 occur in the initialization and record-validation modules, and a `WRITE` occurs in each of the 18 logical switch branches.

The naive service-identification approach would generate 62 different services, one for each `WRITE` statement, without considering whether some of the statements (e.g., the ones that occur in the region of the same tag condition) can be grouped. Thus, for instance, there are eight `WRITE` statements in the `Write-statistics` module; the naive scheme would generate eight backward slices and, thereby, identify eight services, one for each slice.

Table 2 shows some of the services identified by our approach, along with the seed, the bounding region, and a description of the service; the seed, the bounding region, and the identified services are shown by the line numbers. Figure 5 shows the identified services visually to illustrate the service hierarchy. Our approach identifies the 18 tag conditions corresponding to the 18 logical switch-case statements. For each of these tag conditions, our approach finds two service variants by varying the bounding region. The

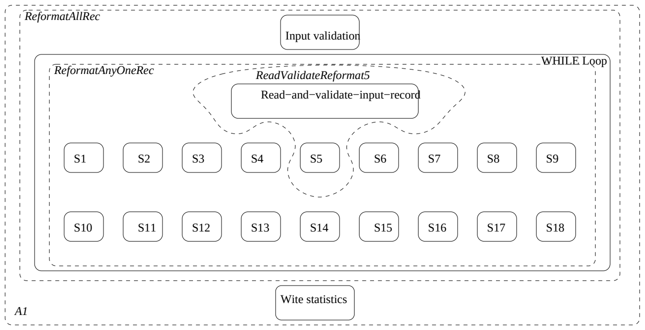

Figure 5: Hierarchical services identified for Benchmark A1. Dashed boxes indicate some of the identified services.

```
Initialization();
while (...) {
   Read-and-validate-input-customer-record();
   while (...) {
     Read-and-validate-activity-record();
   }
}
Write-statistics();
```

Figure 6: Skeleton code of Benchmark A2.

first service is contained in the main loop, which validates the input record and invokes the core service; service 1 in Table 2 is an instance of such a service. In Figure 5, an example of such a service is shown as the dashed curved region labeled *ReadValidateReformat5*. The second service invokes the initialization code, and then performs the core service in a loop; service 2 in Table 2 is an instance of such a service. For lack of space, we do not show an instance of this service in Figure 5.

In addition to these 36 services, our approach finds three more services. The first service (service 37 in Table 2) corresponds to the complete loop body—shown by the dashed rectangle labeled *ReformatAnyOneRec* in Figure 5. The second service (service 38) corresponds to program initialization and the complete loop—shown by the dashed rectangle labeled *ReformatAllRec* in Figure 5. The third service (service 39), which corresponds to the entire program and includes the initialization code, the complete loop, and the code that print statistics, is shown by the dashed rectangle labeled *A1* in Figure 5. The first among these three services is contained within the second one, and further the second one is contained within the third one. The total number of identified services is 39.

An interesting point to node is that although A1 has many more regions than we discuss here—it has 106 tags and 5 loops)—our approach ignores many of them because they do not control any `WRITE` statements. (This is true across all the three benchmarks.) Another noteworthy point is that A1 has loop-carried dependences introduced by different counters, which are part of a single record variable, and are used to collect statistics. The service corresponding to the whole program is identified by specifying these output statements along with the other output statements as the seed.

Benchmark A2

Lines = 1296; *WRITEs* = 19; *Loops* = 2; *Marked global variables* = 0; *Identified services* = 5.

A skeleton of this benchmark program is shown in Figure 6. This program performs the nightly processing of the data of personal banking customers that have been updated during the day. It has a batch loop, which iterates over all the customers; nested within that loop is another loop, which iterates over the activities of each

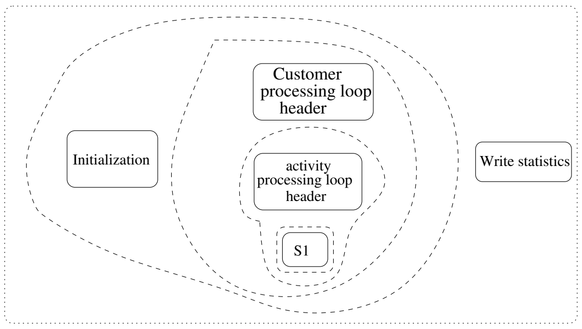
Customer processing loop header
Initialization
activity processing loop header
Write statistics
S1

Figure 7: Hierarchical services identified for Benchmark A2.

customer. The body of the inner loop intuitively defines a service that processes customer data. Of the 19 `WRITE` statements, 16 occur in `Write-statistics` module, one occurs in the initialization module, and the remaining two occur in the body of the inner loop.

The naive service-identification approach would identify 19 services, one for each `WRITE` statement, without considering whether the writes can be grouped. In contrast, our approach identifies only one service corresponding to the inner data-processing code; this is shown as the dashed rectangle S1 in Figure 7.

Although this service is within two nested regions, our approach identifies only one service, and three variants of this core service. The first variant corresponds to the core service invoked within the inner loop (which processes the data for a customer); it is shown by the dashed curved region containing S1 and activity-processing loop in Figure 7. The second service corresponds to the outer loop (which processes the data for all customers); it is shown by the dashed curved region containing S1, the activity-processing loop, and the customer-processing loop in Figure 7. The third service invokes the initialization code before invoking the outer loop; it shown by the dashed curved region containing S1, the activity-processing loop, the customer-processing loop, and the initialization code.

Our approach identifies two more services by using the inner and outer loop bodies as the bounding regions, but these are identical to the services already identified. Finally, our approach identifies another service consisting of initialization of data and the outermost loop, and the last one consisting initialization, whole data processing, and writing of statistics (shown by the dotted rectangle in Figure 7).

Benchmark A3

Number of lines = 803; *WRITEs* = 1; *Loops* = 1; *Marked global variables* = 1; *Identified services* = 2.

A skeleton of this benchmark program is shown in Figure 8. This program contains simple record-processing code, where each record is read in a loop and processed. Two interesting aspects of this program are the following: (1) iteration i of the loop generates the output record based on the input record read in iteration $i - 1$ ($i > 0$), or the record read before the beginning of the loop ($i = 0$); (2) each loop iteration copies the updated statistics to two fields of the output record before the output record is written out.

The naive service-identification approach would generate a single service for the `WRITE` statement. If no variable is marked as global, our technique also identifies only one service (identical to service identified by the naive approach). However, if we identify the variable used to collect the statistics as a global variable, our technique identifies a service corresponding to the part of the

```
Initialization();
Read-record();
while (...) {
   Process-record();
   Update-statistics();
   Copy-statistics-to-output-record();
   Write-output-record();
   Read-record(); }
```

Figure 8: Skeleton code of Benchmark A3.

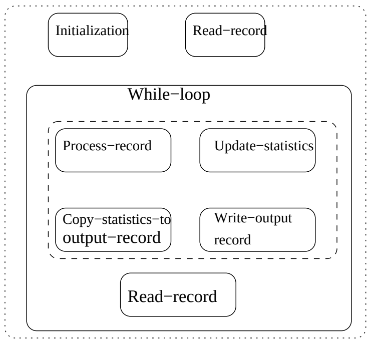

Figure 9: Hierarchical services identified for Benchmark A3.

loop body (shown in the dashed rectangle in Figure 9). Moreover, our technique identifies another service corresponding to the whole program (shown in the dotted rectangle in Figure 9). Note that we do not generate a service that does the initialization and executes an iteration of the loop.

Discussion

Although preliminary, this study illustrates the benefit of our technique that, unlike the naive technique, it does not identify overly fine-grained services. By grouping relevant `WRITE` statements in the same service, our technique identifies meaningful services that are also organized in a hierarchical manner, which arguably can help in program understanding. Moreover, as the data for Benchmark A3 illustrate, our approach can identify services that the naive technique misses. Thus, our results indicate that the approach has the potential to be highly effective in identifying useful services from legacy batch applications.

Another noteworthy point is that the user input (marking the global variables) required by our technique to identify services effectively is minimal for the three benchmarks: of the three benchmarks, only one required global variables to be marked and, even in that case, only one record variable had to be marked.

From our experience in studying Cobol programs, we observe that our modeling of the tag condition has been precise enough. All the tag conditions we have seen in the programs have been of the form (*Field op Const*), where *Field* is a field in a record, $op \in \{=, \neq\}$ and *Const* is a constant literal. For all our subject programs, the inferred services matched the underlying intuitive services.

Although our results are promising, they are preliminary, and further empirical investigation is required to confirm them. The most significant threat to the validity of our observations are threats to external validity, which arise when the observed results cannot be generalized to other experimental setups. In our study, we used only three Cobol subjects of medium size. The effectiveness of the technique may vary for other subjects. However, the three subjects contain typical idioms of batch-processing legacy Cobol applications. Therefore, we are confident that our results may generalize to other subjects as well. An implementation of the approach will let us conduct more extensive empirical studies in the future.

7. RELATED WORK

There has been a significant amount of past work in the area of identifying candidate services from legacy software. These include techniques based on software clustering [25, 26], graph analysis [19], slicing based on programmer specifications of the slicing criterion [6, 11, 18], as well as combined static and dynamic analysis [7, 24]. The approaches in the first two categories mentioned above are fully automated, whereas the latter ones need programmer involvement (in terms of slicing criteria or test cases). To our knowledge, we are the first to provide a formal characterization of a service, based on input/output characteristics and data-flow and control-flow properties, and the first to propose the notion of hierarchical organization of services. Furthermore, we provide a solution that is tailored to the idioms that occur frequently in batch-processing business applications. Finally, although the candidate services we identify are influenced by programmer specifications of global variables, providing these specifications requires less extensive developer interaction than providing slicing criteria for each service, or a good set of test cases.

Hess [12] proposes an approach for categorizing entire programs (not statements) within a batch application based on the system-level idioms and relationships they participate in. Structural transformations of batch applications are also discussed.

There have been many automated as well as semi-automated approaches reported in the literature for identifying related procedures, classes, modules, etc. [2, 4, 5, 23]. These approaches do not address the problem of precise identification of interleaved features.

There has been foundational work reported in the areas of parametric [9] and conditioned [10] program slicing, which integrate partial evaluation and program slicing in a systematic and formal way. It would be interesting future work to use these ideas to provide a characterization of services that is less code-oriented, and that relates the *operational semantics* of a service to the semantics of the containing service or program.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented an automated technique for identifying interleaved, independent services from batch-processing business applications. We presented a characterization of services in terms of input/output characteristics, control-flow properties, and data-flow properties. We then presented an algorithm that combines tag-condition analysis with program slicing to identify services that conform to our characterization. Our preliminary empirical results indicate the potential usefulness of the approach.

Implementation of the proposed technique and a detailed evaluation remains the main future work. There are several problems in the area of service identification and extraction that can make for challenging future research. For example, it would be interesting to formalize the quality or usefulness of an identified service. Similarly, approaches to filter and rank identified services would be useful. More powerful transformation techniques (e.g., making use of program-rewriting techniques) ought be explored to expand the idioms of batch programs that can be addressed.

9. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, Boston, MA, USA, 1986.

[2] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. A method to re-organize legacy systems via concept analysis. In *Proc. Int. Workshop on Program Compr. (IWPC)*, pages 281–290, 2001.

[3] T. Ball and S. Horwitz. Slicing programs with arbitrary control flow. *Lecture Notes in Computer Science*, (749):206–222, 1993.

[4] G. Caldiera and V. R. Basili. Identifying and qualifying reusable software components. *IEEE Computer*, 24(2):61–70, 1991.

[5] F. Chen, S. Li, and W. C.-C. Chu. Feature analysis for service-oriented reengineering. In *Proc. 12th Asia-Pacific Software Engineering Conference*, pages 201–208, 2005.

[6] A. Cimitile, A. D. Lucia, and M. Munro. Identifying reusable functions using specification driven program slicing: a case study. In *Proc. Int. Conf. on Softw. Maint. (ICSM)*, pages 124–133, 1995.

[7] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, 29(3):210–224, 2003.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Sys.*, 9(3):319–349, July 1987.

[9] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *Proc. Symp. on Principles of Progr. Langs.*, pages 379–392, 1995.

[10] C. Fox, S. Danicic, M. Harman, and R. M. Hierons. Consit: a fully automated conditioned program slicer. *Softw. Pract. Exper.*, 34(1):15–46, 2004.

[11] M. Harman, N. Gold, R. M. Hierons, and D. Binkley. Code Extraction Algorithms which Unify Slicing and Concept Assignment. In *Working Conf. in Reverse Engg. (WCRE)*, pages 11–21, 2002.

[12] H. M. Hess. Aligning technology and business: Applying patterns for legacy transformation. *IBM Systems Journal*, 44(1):25–46, 2005.

[13] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *Proc. Int. Workshop on Program Comprehension*, pages 33–42, 2003.

[14] R. Komondoor and G. Ramalingam. Recovering data models via guarded dependences. In *Proc. Working Conf. on Reverse Engg.*, pages 110–119, 2007.

[15] K. Kontogiannis, G. Lewis, , and D. Smith. A research agenda for service-oriented architecture: Research needs for maintenance and evolution of service-oriented systems. In *Proc. 2nd Intl. Workshop on SOA-Based Systems Maint. and Evolution (SOAM), 12th European Conf. on Softw. Maint. and Reengineering (CSMR)*, 2008.

[16] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proc. Symp. on Principles of Progr. Langs.*, pages 207–218, 1981.

[17] A. Lakhotia and J. Deprez. Restructuring programs by tucking statements into functions. *Inf. and Softw. Technology*, 40(11-12):677–689, Nov. 1998.

[18] F. Lanubile and G. Visaggio. Function recovery based on program slicing. In *Proc. Conf. on Software Maint. (ICSM)*, pages 396–404, 1993.

[19] S. Li and L. Tahvildari. A service-oriented componentization framework for java software systems. *Proc. Working Conf. on Reverse Engg.*, pages 115–124, 2006.

[20] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proc. ACM Softw. Engg. Symp. on Practical Softw. Development Environments*, pages 177–184, 1984.

[21] V. Rajlich and N. Wilde. The Role of Concepts in Program Comprehension. In *Int. Workshop on Program Compr. (IWPC)*, pages 271–280, 2002.

[22] S. Rugaber, K. Stirewalt, and L. Wills. Understanding interleaved code. *Automated Software Engg.*, 3(1-2):47–76, June 1996.

[23] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proc. 21st Intl. Conf. on Softw. Engg.*, pages 246–255. IEEE Computer Society Press, 1999.

[24] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, 1995.

[25] Z. Zhang, R. Liu, and H. Yang. Service identification and packaging in service oriented reengineering. In *Proc. 17th Int. Conf. Softw. Engg. and Knowledge Engg. (SEKE)*, pages 14–16, 2005.

[26] Z. Zhang and H. Yang. Incubating services in legacy systems for architectural migration. In *Proc. Asia-Pacific Software Engineering Conference (APSEC)*, pages 196–203, 2004.