RI 09001, 1 January 2009 Computer science

IBM Research Report

Using slicing to extract online services from batch programs

Raghavan Komondoor
Department of CSA,
Indian Institute of Science
Bangalore, Karnataka, India

V. Krishna Nandivada
IBM, India Research Lab,
Embassy Golf Links Business park,
Block D, Floor 3,
Bangalore, Karntaka, India

Saurabh S Sinha
IBM, India Research Lab,
Plot No 4, Phase II, Block – C
ISID Campus, Instituional Area, Vasant Kunj
New Delhi, India

John Field
IBM T J Watson research center
P.O. Box 704
Yorktown Heights,
NY 10598, USA

IBM Research Division
Bangalore - Delhi - T.J. Watson

Abstract. Transaction processing a key constituent of the IT workload for commercial enterprises (e.g., banks, insurance companies). Even today, in many large enterprises transaction processing is done by legacy batch applications, that run offline and process accumulated transactions. Identifying independent services from these batch applications is a hard problem the enterprises are grappling with, and one without satisfactory automated solutions.

In this paper we propose a novel static-analysis-based solution to the problem of identifying transaction processing loops in batch programs that can bedeserialized, to yield services that are invocable online. Our solution is based on a combination of detecting conditional code execution regions, backward slicing, and parallelizability detection, and is well suited to the idioms commonly exhibited by batch programs. We evaluate our solution in the context of a case study.

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). A lot of these applications are written for execution in "batch" mode; that is, the application runs periodically (according to some schedule), rather than be available "online" all the time. Whenever a business event occurs (e.g., a customer places a new order), the event is is turned into a "transaction" (input file record) and batched up. During each run of the application all transactions that have been batched up since the previous run are processed sequentially.

While batch processing was suitable for the business environment of the previous decades, today's customers demand online processing of events, and today's computing infrastructure allows this. As a result many enterprises are in the midst of a largely manual and labor intensive effort to modernize and migrate their monolithic legacy applications, in particular to a Service Oriented Architecture (SOA)[1]. In this paper we address the problem of reengineering batch applications that do offline transaction processing, to automatically extract core business logic inside them as "services" invocable online. The extracted services can be invoked online as and when business events occur; they will receive input file records as parameters (this corresponds to execution of input statements in the batch code), transform input records to output records, perform any necessary updates to persistent stores (as in the batch program), and finally return the output records (corresponding to the execution of output statements in the batch code). In other words, we address the problem of "deserializing" offline transaction-processing batch programs. (We do not address non-transaction-processing batch programs, e.g., ones performing analysis, aggregation, reconciliation, and reporting on persistent data.)

[1] a particular style of distributed online computing.

1.1 Motivating example

Consider the order processing batch program in Figure 1, which we use as a running example in this paper. It is written in a simplified variant of the Cobol language, which is the predominant language of legacy batch 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. Cobol has variable declarations, but no user-defined type declarations; also, 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 `PROCEDURE DIVISION`. The program has a main loop, in lines 8 through 46. The loop processes a single transaction (an order) from the input file in each iteration. Each transaction consists of an order "header" record (read into variable `order-rec` in the program), followed by a number of item records, one for each item that's 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 kinds of orders: "regular" orders, handled in lines 13 through 24a ("then" branch of the "if" conditional in line 13), and "full" orders, handled in lines 25 through 44 ("else" branch). Both kinds need an inner loop to process the multiple items that are part of a single order. The program processes each order (described below) and writes it out to the file `out-file`.

This program is typical of batch programs, which usually have a loop that reads transaction records from an input file, and "dispatch" each transaction to an appropriate region of code within the loop body depending on the kind of transaction. The code region applies some business logic on the transaction, creates one or more output records, and writes them out.

A naive solution to service identification would be to apply backward slicing from each output statement. In the following part we discuss using our running example the candidate service fragments identified by our approach, and why the naive solution is insufficient.

Candidate code fragments identified by our approach Each set of statements in Figure 1 labeled by the same number (e.g., (2)) form a candidate service fragment identified by our approach.

Code labeled (2). This forms the *body* of the "regular" orders inner loop. This code considers the current item within the current order (read into variable `in-rec` in line 17), reduces the quantity of the item in the order if there are not enough available in the inventory (as per table `item-table`), and writes out the adjusted item record to the output file.

Note that our approach identifies only the inner loop body (i.e., a single iteration), and not the entire inner loop, as the candidate fragment for the service; intuitively, it has "deserialized" the loop because it is able to infer that processing any individual item in a regular order has nothing to do with the other items in the same order. If our approach had not used loop iteration independence analysis, and instead only backward slicing from the `WRITE` statement in line 23,

we would have included the entire inner loop (i.e., lines 16a through 24a), and even several statements before this loop, as part of the candidate fragment.

Code labeled (3). A "full" order is one that cannot be adjusted. If all the items are available in the quantities requested then the entire order is written out (unchanged), else the entire order is ignored (not written out). Therefore, the program has two inner loops to process a full order – one in lines 27 through 35 to check whether the order can be fulfilled, and the other in lines 40 through 42a to write out a fulfilled order.

Our approach infers that no individual item can be processed in isolation, and that all items need to be processed together; therefore, it identifies the code sequence in lines 25 through 44, including both inner loops (but excluding lines 37 and 38) as the candidate service[2].

Note: Even though both inner loops are included in the candidate fragment, line 37, which is in between the two loops, and which is not pertinent to the logic for processing an individual order (it increments the count of fulfilled orders), is excluded from the fragment. Backward slicing is necessary for eliminating such extraneous code from candidate services.

Code labeled (1, 4) These two fragments write out to the output file, respectively, the "header" record of each regular order, and the header record of each full order. The procedure extraction algorithm invoked at the end of the algorithm, after candidate fragments have been identified by the algorithm, puts back fragment (4) as a part of fragment (3), but leaves fragment (1) as a separately callable service.

For the service identification work to be useful, these code fragments need to be extracted into individual services (functions). Procedure extraction is not a key contribution of of this work, and hence we defer discussion of it to the Appendix).

Note that we have not identified any services containing output statements in lines 5 and 47. These are "header" and "trailer" output statements, not present in any loop in the batch program. These output statements are artifacts of the batch program. They do not perform functionality that is meaningful for online callers of services, and hence we ignore such statements in this paper for the purposes of service identification.

1.2 Our Approach and Contributions

Identifying candidate code fragments for extraction into services and presenting them to the user (as illustrated in Figure 1) is the primary goal of our approach. Our approach is based on three static analyses: tag condition analysis, backward slicing, and loop iteration independence analysis. In summary, we

1. Identify key loops. These loops could be outer loops or inner loops.

[2] The program uses an array `ord-item-recs` of (arbitrarily) bounded length 100 to store the item records in memory between the two inner loops. Typical Cobol programs do not use dynamic memory allocation, and so sometimes do impose such arbitrary restrictions.

2. Within each such loop identify, using a dataflow analysis, "tag" conditionals that demarcate different regions of the loop body that process different kinds of transactions (e.g., regular orders vs. full orders).
3. Classify the loops as:
   - loops whose iterations are independent. The service code in this case contains (a part of) the loop body, but not the *entire* loop. Basically, *each* iteration of the loop corresponds to an invocation of the service.
   - loops whose iterations are dependent on each other. The entire loop, plus potentially other computations that precede the loop that the loop depends on, constitute the service.
4. In either case above, we use a variant of the *backward slicing* [1] operation to identify the statements that constitute the service – the statements that transform the input records to the output records.
5. Extract the identified code fragments as separate procedures, which could be wrapped as services. For this we leverage our previously reported algorithm for procedure extraction [2].

The key contributions of this paper over previous related work are that

- To our knowledge, ours is the first automated static-analysis based approach that specifically addresses the offline transaction processing idiom,
- We use a combination of three static analysis techniques, namely, tag condition analysis, backward slicing, and loop iteration independence analysis, to address the above-mentioned idiom satisfactorily. As illustrated by our running example, individual techniques by themselves are insufficient in general.
- We engineer our analysis to produce extractable code fragments, and then "go the last mile" by linking the output of our analysis to a procedure extraction algorithm.

1.3 Related Work

Previous related approaches we know of do not specifically address the task of deserializing loops in a transaction processing batch application. They are generally in the space of modularization of monolithic applications, and fall into the following categories:

- A user somehow specifies, using an artifact external to the program's code, the functionality they are looking for. For example, using a slicing criteria [3, 4], test cases [5], or use cases [6].
- A large number of automated approaches for clustering of related procedures, e.g., [7, 8], and using Feature Oriented Analysis [9].

These techniques in general do not take the loop parallelization techniques into account that are useful in the context of our problem.

Note that while the batch applications we target are a category of monolithic applications (because they process different kinds of transactions, and perform

various different tasks), they commonly employ a specific, unique idiom of their own, which we leverage to provide a tailored solution.

Many previous publications provide informal, or mostly manual, or structural approaches to addressing the batch idiom, and discuss industrial case studies. For example, Hess [10] proposes an approach to categorizing code at the program level (not statement level), and uses a library of structural patterns to identify instances of system-level idioms (i.e., at the level of programs and their relationships), and to transform these instances.

We borrow the ideas of loop-carried dependences that parallelizing compilers have used for a long time [11]. Our area of application is different, and hence the idioms are different, too. On a different note, there has been foundational work reported in the areas of parametric [12] and conditioned [13] program slicing, that we could leverage to extend the power of our approach.

Finally, there is work in the area of automated migration of network database traversal loops in Cobol applications to SQL queries [14]. Their approach uses a plan-calculus representation of programs, and a pattern-database to find and rewrite patterns in the plan-calculus representation iteratively until loops are replaced by SQL queries. Although our problem and our technique is different from theirs, both approaches, in effect, look for loops with independent iterations. Their transformation raises the level of abstraction of code more than ours, but at the same time is applicable mainly to database-traversal loops, which are more stylized and regular than transaction-processing loops.

2 Assumptions and terminology

For our algorithm we assume a simple imperative language, with the usual constructs such as assignments, sequencing, conditional statements, loops, jumps, `READ` statements (input statements, both sequential and keyed), `WRITE` statements (output statements, both sequential and keyed), and procedure calls. Variable declarations are as in Cobol: "*var* `PIC X(`n`)`" or "*var* `PIC 9(`n`)`" declares the variable *var* to store a byte sequence of length n, where `X` means the bytes store arbitrary characters, while `9` means the bytes store decimal digits. Conditional statements are assumed to be side-effect free; if a conditional does have side effects it ought to be first transformed to remove the portions with side effects into separate statements. I/O statements have the standard semantics, which we illustrate using the example program in Figure 1: (a) "`READ in-file`" reads the next data record from the sequential input file `in-file` into the associated variable `in-rec`, (b) "`READ item-table KEY IS` *val*" reads the record from the random access database table `item-table` that has value *val* in the primary key field `item-id`, while (c) "`WRITE` *val* `TO out-file`" appends the (scalar, or record-structured) value *val* to the sequential output file `out-file`.

We assume that (conservative) dataflow dependences [15] between statements have been pre-computed. This includes a standard inter-procedural analysis to compute summaries for procedure calls, where a summary of a procedure call is a set of variables (global variables, and variables in the caller) that may be

referenced and a set of variables and that be defined as a result of the procedure call. Dataflow dependences from output statements on a random-access file to input statements on the same file are assumed to have been pre-computed.

We also assume that control dependences [16] between statements have been pre-computed; intuitively, a statement is control dependent on a conditional or loop header if the conditional or loop header determines whether and how many times the statement executes. 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.

Regarding our notation, a *node* is a statement, or conditional, or loop header. If l is a loop header node (e.g., a `while` conditional), then $loop(l)$ identifies the set of all nodes inside the loop headed by l, including l itself. $\mathcal{LH}$ is the set of all loop headers in the program. $m \xrightarrow{f} n$ indicates that node n is dataflow dependent on node m. The construct $p \xrightarrow{c} n$ indicates that node n is directly or transitively control dependent on either branch edge out of the conditional p. $dom(p \xrightarrow{b}, n)$ indicates that the edge labeled b (b is either *true* or *false*) out of conditional p *dominates* node n (i.e., all paths from the program entry to n go through the aforementioned edge). A backward slice [1] of a program from a statement s is the set of statements on which s is dependent by the transitive closure of the dataflow and control-dependence relations.

3 Algorithm description

3.1 The idioms we address

We address transaction processing batch programs that follow certain idioms, which are quite common in practice. (The approach will process all programs, but the results obtained may not be as good if they follow different idioms.) We assume that each data file is accessed by the program either (a) sequentially, in read-only mode (i.e., this is an input/transaction file), or (b) using append operations only (this is an output file), or (c) as a "master" file, in random access mode, for reads and/or updates. This classification of the files is an input to the algorithm (although we expect that heuristics, or even sound analysis techniques, could be developed to infer this classification). We only permit keyed accesses for master files; they are not iterated over to access all records in the file.

If a batch program expects the input to be reordered (say sorted) before being fed to it, then such a batch program when translated to a (semantically equivalent) online version [3] may, in the absence of the correctly reordered input, result in undefined behavior. Hence, we assume that the batch program's logic is not contingent upon the input file records being in any sorted or grouped order.

Of the above restrictions, the one about input files being "unsorted" may be the one that is violated more often than others. However, in our observation

[3] The online version of the program cannot expect to have all the input records beforehand, like the batch program.

fairly small (manual) changes to the program are often enough to make them respect this restriction. For example, a sorted input file may need to be turned into a master file, and sequential accesses to it be replaced by random (keyed) accesses. Such a transformation is necessary in any case for the program to be transformed into an online collection of services; providing automated support for this transformation is left to future work.

3.2 The algorithm

Figure 2 provides an outline of the steps in our algorithm. Note that any single candidate fragment we identify is contained within a single procedure; this fragment could contain procedure calls, including calls to other procedures we have identified and extracted, but not *portions* of multiple procedures. Extending the approach to the general case would involve using inter-procedural slicing to find the fragments, and devising some form of inter-procedural code extraction. However, our technique may be sufficient for Cobol, which does not support recursion, and wherein it may be feasible to fall back on inlining to eliminate all procedure calls. The remainder of this section presents the details of the steps in the algorithm, while Section 3.3 has a qualitative discussion on some of these steps.

Step 1. Identify tag fields and conditionals in the program A *tag field* is intuitively a field in an input record that helps identify what kind of transaction it is. In this paper we characterize a field $r.t$ of an input record r read at some input statement s as a tag field if the value $r.t$ may reside in a variable v (reaching v via zero or more copy statements) at a program point wherein there is an "if" conditional p such that some sub-condition within this condition is a comparison of v with some literal c using a "$=$" or "$\neq$" operator. We also say in this case that p is a *tag conditional*, c is a *tag literal associated* with tag field $r.t$, and variable v in conditional p *stores* tag field $r.t$ (written as $(v = r.t)@p$ or $(v \neq r.t)@p$).

In the running example in Figure 1, where `order-rec` stores an input (order header) record, the `ord-type` field within it is a tag field; this field is compared against literals in lines 13 and 25, and is used to identify the order as a regular order or full order. Note that often tag conditionals control the different parts of the program that process different kinds of transactions.

In our observation our characterization identifies almost all tag conditionals in real programs. It sometimes (over)characterizes certain non-tag conditionals as tag conditionals, but Step 4 of our algorithm compensates for this by heuristically filtering away some of these mis-characterized conditionals. It may be noted that, over-characterization may impact the usability of the service, but not the semantic correctness aspect of it.

In weakly-typed languages such as Cobol, tag fields and tag conditionals are not explicit in the program. Moreover, in general, a tag field value may be moved to a temporary variable (via one or more assignment statements, transitively) before being tested in a tag conditional. Therefore a transitive data dependence computation is necessary to discover what portions (fields) of input records flow

into what variables referred to in program conditionals. A dataflow analysis approach we reported earlier [17] is an instance of such a computation for weakly-typed languages such as Cobol; we use this analysis in Step 1 of our approach, in conjunction with the characterization we propose above, to identify tag fields, their associated tag literals, the *stores* relation (see above), and tag conditionals.

Step 2. Identify seeds for backward slicing A service invocation ends with one or more output records being returned to the invoker. This, in the original batch program, corresponds to the execution of one or more `WRITE` statements (to output files, or to master files). Therefore, our approach is to start at `WRITE` statements, and to slice backward to find code that creates the records written at these statements. The question is: what set of `WRITE` statements in a batch program ought to be grouped as a *single* slicing criteria, meaning we would treat the union of the backward slices from all `WRITE` statements in the group as a single candidate service?

Preliminaries. We introduce a procedure *pred* such that for any "if" conditional p in the program and boolean value b ($b = true$ or $false$), $pred(p \xrightarrow{b})$ is a boolean formula on tag fields and literal values that *must* be true when p evaluates to b. We do not provide a formal definition for this procedure due to lack of space; it is a syntax-directed translation of the conditional expression in p. We provide three examples:

$$\frac{p{=}\texttt{if(v=1 AND w=2)},\ (\texttt{v}{=}r.t1)@p,\ (\texttt{w}{=}s.t2)@p}{pred(p^{true}_{\rightarrow}) =_{def} ((r.t1{=}1){\wedge}(s.t2{=}2))} \qquad \frac{p{=}\texttt{if(v=1 OR w=2)},\ (\texttt{v}{=}r.t1)@p,\ (\texttt{w}{=}s.t2)@p}{pred(p^{true}_{\rightarrow}) =_{def} ((r.t1{=}1){\vee}(s.t2{=}2))}$$

$$\frac{p{=}\texttt{if(v=1 AND w=2)},\ (\texttt{v}{=}r.t1)@p,\ \texttt{w}\text{ does not store any tag field}}{pred(p^{false}_{\rightarrow}) =_{def} (r.t1{\neq}1)}$$

For any statement s, we define

$$pred(s) =_{def} \bigwedge_{p \mid dom(p^{b}_{\rightarrow}, s)} pred(p \xrightarrow{b})$$

Intuitively, $pred(s)$ indicates the literal values that tag fields *must* take whenever control reaches s[4].

Approach. We call any maximal set S of `WRITE` statements (writing to output files or master files) a "seed" and treated it as a single slicing criteria if the following two properties hold:

- All statements in S are contained in the same procedure, and within this procedure, within the same set of loops.
- For any two `WRITE` statements s_1 and s_2 in S the boolean formula $pred(s_1)$ is equivalent to the boolean formula $pred(s_2)$.

[4] We treat the entry node of the program as a pseudo-predicate, and treat the *true* edge out of the entry node as dominating all nodes in the program.

The intuition behind the first restriction above is that each loop should potentially be able to yield its own service. The intuition behind the second restriction is that a set of `WRITE` statements that are all involved in the processing of the same kinds of transactions ought to yield a single service. Note that even though checking equivalence of boolean formulas in general is in co-NP, we can sidestep this complexity issue by restricting our formulas, e.g., to 2-SAT (at the expense of precision).

Note that Step 2 is inside the main loop in the algorithm (Figure 2). If there are multiple seeds available to pick in any iteration, Step 2 picks one of them, and leaves the rest for processing in subsequent iterations. The heuristic we propose to select among these seeds is to pick the one that appears earliest in a topological ordering of the seeds in the control-flow graph of the program (with ties between seeds resolved arbitrarily). The intuition behind this is that if the backward slices from two different seeds overlap, it is usually a good idea to first identify and extract (into a separate procedure) the slice that occurs "earlier" in the program, and then, on the thus transformed program, identify and extract the "later" slice (which could include the call to the earlier extracted function).

Illustration. Consider the running example in Figure 1. The seeds available during the first iteration of the algorithm, in a topological order, are: {5}, {15}, {23}, {38}, {41}, and {47}. In this example none of the seeds happens to contain more than `WRITE` statement.

Step 3. Identify an initial candidate fragment

Figure 3 describes the procedure for growing a seed S into an "initial" candidate fragment (i.e., set of nodes) $f_{init}(S)$ using a variant of backward slicing. The initial `while` loop in the figure simply implements backward dataflow-only slicing, while not going past input statements. The subsequent `forall` loop identifies loops l that contain a cycle of dataflow dependence edges all of which were traversed during the slicing, such that there is no other loop nested inside l that contains this cycle. Such loops are characterized as ones whose iterations are dependent on each other, and added in their entirety to $f_{init}(S)$.

Illustration. Consider the seed that contains the `WRITE` statement in line 41. The backward dataflow slice from this statement, excluding input statements, consists of

$f_{init}(\{41\})$ contains lines 26 through 43, excluding lines 37 and 38, plus line 12. Let us see how. These are the statements that lie on the backward dataflow slice from statement 41. The `READ` statement in line 9 also lies on the backward slice, but is not added to $f_{init}(\{41\})$ (see the **while** condition in the algorithm). There are no cycles of dataflow dependences encountered during the slice operation, so the **for all** loop in the algorithm does nothing. □

Note, we do not necessarily characterize a loop that has loop-carried dependences [15] as one whose iterations are dependent on each other. To see why, consider the (fairly common) scenario of a "`do while`" loop with a `READ` statement just before the loop to obtain an input record for the first iteration, and another `READ` statement at the end of the loop body to obtain the input record

for the next iteration. In this scenario there are loop-carried dataflows, and they would even be traversed while slicing back from statements in the loop that use the input record. However, assuming there are no other dataflows from (non-input) statements in one iteration to the statements in subsequent iterations, this loop intuitively has independent iterations, in the sense that the *processing* performed in one iteration does not influence the processing performed in the next. In other words, this loop is semantically equivalent to a "`while do`" loop that has a `READ` statement at the beginning of the loop body, which has no loop-carried dependences. It is to treat these two loops equivalently for the purposes of identifying candidate service fragments that we look for a full *cycle* of dataflow dependences (which would naturally include a loop-carried dependence).

Note that while we have a satisfactory approach to identifying independent iterations, the automated extraction algorithm we use (in Step 5) does not do an ideal job when applied to the body of the "`do while`" mentioned above. Ideally, this loop should be rewritten as a `while` loop prior to the extraction; such a transformation would move the `READ` statement in the loop body to the beginning of the loop body, break the loop-carried flows, and hence eliminate the need to pass input records read in one iteration to subsequent iterations (i.e., after service extraction, from one call to the service to the next call). Currently, we leave such transformations to be done via manual intervention.

Step 4. Pruning the initial candidate fragment $f_{init}(S)$ Previous work on automated procedure extraction, suggests that the first step in extracting a (possibly non-contiguous) fragment of code into a separate procedure is to identify a *hammock* [18] or an *e-hammock* [2] in the program that bounds the code to be extracted. In this paper, we use the notion of an e-hammock; an e-hammock corresponds to a sequence of statements (simple or compound) in the program such that there are no outside jumps whose targets are inside the sequence (excluding the entry node of the sequence).

Our goal in this step is to identify an e-hammock $\mathcal{H}(S)$ that bounds the initial candidate fragment $f_{init}(S)$ identified in the previous step. As part of this, we may prune away certain nodes from $f_{init}(S)$ that cannot be accommodated without making the bounding e-hammock so large that it violates certain characteristics we desire. The details of this are presented in Figure 4. Basically, we first identify a "core" subset $f_{core}(S)$ of the initial set of nodes $f_{init}(S)$. Then, we identify $\mathcal{H}(S)$ as an e-hammock that (a) bounds the nodes in $f_{core}(S)$, (b) bounds *as many* of the non-core nodes in $f_{init}(S)$ *as possible*, (c) *excludes* all tag-conditionals that are not in the core subset but control nodes in the core subset, and (d) excludes loop headers that are not in the core subset but whose bodies contain core nodes.

The "core" nodes consist of S itself, plus any loops that contain S that are entirely contained within $f_{init}(S)$. The core nodes are intuitively the ones that definitely need to be present in the extracted service. A loop in $f_{init}(S)$ whose body contains S is considered core because in the presence of loop-carried dependences (which are the cause for an entire loop to be included in $f_{init}(S)$), it does not make sense to extract just a single iteration of the loop, i.e., just the

loop body, into a separate service. Loops that do not contain S are considered non-core, because they can potentially be left behind in the rewritten batch program, before the call to the extracted service.

Illustration. For example, consider the seed statement 41 in Figure 1. As discussed in earlier in this section, $f_{init}(\{41\})$ contains lines 26 through 43, excluding lines 37 and 38, plus line 12. This is essentially the backward data-only slice from statement 41. $f_{core}(\{41\})$, computed in this step, contains only lines 40 through 42a. $\mathcal{H}(\{41\})$ contains lines 26 through 43 (including lines 37 and 38), but excludes line 12.

Step 5. Extracting the candidate service Our final step is to simply invoke the procedure extraction algorithm reported in [2] on the bounding e-hammock $\mathcal{H}(S)$. The extraction algorithm transforms $\mathcal{H}(S)$ to extract the nodes within it that also belong to $f_{init}(S)$ (i.e., the nodes belonging to the candidate fragment) (plus potentially other nodes in $\mathcal{H}(S)$ that it could not move out of the way) into a separate procedure. This step being not a central contribution of this paper, we defer further discussion of it to the Appendix.

The algorithm is semantics preserving The transformation done by our algorithm is guaranteed to be semantics preserving: the original batch program and the rewritten batch program exhibit identical externally observable behavior [19]. We present here an intuitive reasoning for this: (a) Each individual call to the procedure extraction algorithm is semantics preserving (see [19]), and (b) We do *not* identify multiple (potentially overlapping) slices all up front before beginning extraction. Rather, we identify a slice, extract into a procedure, identify another slice on the thus transformed program, extract it, and so on. Therefore, there is no issue of determining, for a statement that is in multiple slices, which one(s) to assign it to.

3.3 Qualitative discussion of the approach

In this section we discuss qualitatively the strengths and limitations of the main steps of the algorithm, and provide more insight into our design decisions.

Discussion of step 2 Note that two `WRITE` statements need not necessarily be in the same basic block of the program to be in the same seed. For example, if there are two different tag conditionals in the program (within the same loop) that both compare the same tag field with the same tag literal using the same comparison operator (say, "="), then the `WRITE` statements in the "`then`" (resp. "`else`") branches of these two conditionals will be in the same seed (provided these statements are not contained in any loops that are inside these branches). In other words, our analysis is able to correlate different parts of the program that pertain to the processing of the same kind of transaction, and identify services that span across these different parts.

Discussion of Step 3

Reason for dataflow-only slicing

We do not follow control-dependence edges backward during slicing, because a "full" backward slice from a statement (traversing both dataflow and control-dependence relations) is likely to yield a much larger candidate fragment than dataflow-only slicing. Our intuition is that dataflow dependences are the ones that relate statements pertinent to processing the same kind of transaction. We take control dependences into account in the Step 4 of the algorithm, after we have found an initial candidate fragment in this step.

Cohesion and coupling

Our approach strives for high cohesion [20] within a candidate fragment simply by using backward dataflow slicing, which intuitively identifies a single thread of computation (conceptual task). As an example, consider that the backward slices from the two `WRITE` statements in the running example (in lines 23 and 41) *do not* include the statements that increment variable `num-fulfilled`. This results in good cohesion, because incrementing this reporting variable is not part of the same conceptual task as the one that determines *how* to process the order. Similarly, we aim for low coupling by *not* deserializing loops that have dependent iterations; e.g., were each item in a full order to be processed by an independent service invocation, there would be high coupling between invocations related to the same order (because either all items have to be accepted, or none).

Input statements on sequential files

We don't normally include input statements on sequential files in the candidate fragments, because these make more sense if they are left behind in the rewritten batch program. A service ought to receive an input record as a parameter, because this mode is more suitable for online usage.

However, if the service needs an unbounded number of input records, as in the case of service (3) in the running example, the "`forall`" loop in Figure 3 kicks in, and places the input statements (actually, the entire loop that contains them) in the candidate fragment. During actual conversion of the extracted procedure into a service, the service invoker fetches input records in the place of `READ` statements.

Discussion of step 4 The most interesting aspect of this step is the restriction in the second last bullet point in Figure 4. Recall that the procedure extraction algorithm in Step 5 always places in the extracted procedure all conditionals and loop headers in the e-hammock $\mathcal{H}(S)$ that control nodes in $f_{init}(S)$. One thing this restriction achieves is to *exclude* a loop header l that is *not* in $f_{core}(S)$ from $\mathcal{H}(S)$. Including l in $\mathcal{H}(S)$ would be undesirable, because then the (entire) loop $loop(l)$ would end up in the extracted procedure. This would defeat the intention of Step 3, which excluded l in the first place from $f_{init}(S)$ because it inferred that the loop $loop(l)$ could be deserialized, and need not be placed in its entirety in the service created for seed S.

Another thing this restriction achieves is to exclude tag conditionals that are not core but that control core nodes. Consider, in the running example, the seed {41}. The tag conditionals in lines 13 and 25 ought *not* to be included in the full-order service, because it is these conditionals that *determine* when the full-order service needs to be called and when the regular order service needs to be called. Rather, these tag conditionals should remain in the rewritten batch program, and should control the calls to the respective extracted services.

The reason we do not exclude conditionals that do not control core nodes is that these conditionals are likely to be not real tag conditionals, and to have been mis-characterized as tag conditionals by Step 1.

4 Evaluation

We have proposed a novel static-analysis based approach to deserializing batch transaction processing programs into a collection of online-invocable services. We have performed an informal, manual evaluation of the techniques on a collection of real (proprietary) programs. For lack of space, we restrict ourselves to presenting our experience of applying our techniques on three medium sized programs (each over 1000 lines long). Each of these batch programs had a single outer most loop, that processed all the input records. Each of the program consisted of a sequence of "Cobol Paragraphs", which are "performed" at different places. For the purpose of this evaluation, we assumed that all the paragraphs were inlined. We discuss our experience with each of the program, followed by some general observations. For each benchmark we present a few statistics: number of lines of Cobol code, number of tag conditions, number of write statements, and the number of identified services.

Benchmark A1: *Number of lines* = 1323; *tags* = 61; *WRITEs* = 18; *Services* = 18. The main loop of this benchmark processes 12 kinds of transactions (controlled by a 12-way switch). Some of these 12 regions had further switches within, because not all fields in the transaction record contained valid data always, with different combinations of fields that could contain valid data. In total, there were 18 switches, including the outer one and inner ones, and our approach identifies 18 services. Both this benchmark and the next one (A2), had loop carried dependencies introduced by different counters present in the loop. However, these counters were not written out to the main output file, thus not introducing any loop carried dependencies between output statements in different iterations. Our techniques recognized this idiom well.

Benchmark A2: *Number of lines* = 1296; *tags* = 8; *WRITEs* = 2; *Services* = 2. The second program had a batch loop, within which two different activities were done, one after the other. In this benchmark, there were two interesting issues: (a) There was some common code that the two write statements, present in two different seeds, were data dependent on. This is in contrast to program A1, where there was no dependency among different seeds. Based on the order in which the seeds are chosen to be extracted out, the code content in the corresponding extracted services would vary. Our suggested topological order

for processing the seeds helped identify the intuitive service segments. (b) there was a 2-way switch processing two different kinds of transactions, but followed by a single `WRITE` statement that writes out the output record coming out of both branches. Our approach identified only one service here. We might be able to extend our approach to use conditioned or parametric slicing, in conjunction with the tag conditionals we infer, to identify two services here.

Benchmark A3: *Number of lines* = 803; *tags* = 1; *WRITEs* = 4; *Services* = 2. This program had a batch loop whose body first processed the transaction to produce an output record written to an output file, and then, if there was an error condition, generated an additional (error) record in an error file. The first one (includes two write statements) involved writing to the output, the current iteration number as well. This introduced loop carried dependency among this specific write statements and the whole of the outer loop (including the loop header) resulting in the whole of the batch loop being considered as a service. While processing the second seed (consisting of the other two write statements), our techniques carved out another service as a sub-service of the first one.

Other Observations

• From our experience in studying Cobol programs, we can say that our modelling 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 some constant literal.

• In general there can be a tie in the order in which seeds are picked for processing in step 2. In such cases, some sort of user intervention might be expected. However, during our study, we did not encounter such a situation.

• Our experience in applying these techniques to the above examples showed the importance of our proposed five step approach. Overall, we felt that all but one (See discussion on benchmark A2, point (b)) of the services identified by the approach were quite close to the set of services recognized by us.

5 Future Work

Our work can be seen as a step towards integrating the capabilities of different program analysis techniques to realize independent services from batch programs. There are several issues in this area that can make for challenging further research. Coming up with a formalization of "quality" for the discovered services in the online context would be a significant step towards evaluating any service-identification algorithm. Also, more powerful transformation techniques (e.g., making use of program rewriting techniques) could be explored to expand the idioms of batch programs that can be addressed.

References

1. Weiser, M.: Program slicing. IEEE Trans. Soft. Engg. **10**(4) (1984) 352–357
2. Komondoor, R., Horwitz, S.: Effective, automatic procedure extraction. In: Proc. Int. Workshop on Program Comprehension. (2003) 33–42

3. Lanubile, F., Visaggio, G.: Function recovery based on program slicing. In: Proc. Conf. on Software Maintenance. (1993) 396–404
4. Cimitile, A., Lucia, A.D., Munro, M.: Identifying reusable functions using specification driven program slicing: a case study. In: Proc. Int. Conf. on Softw. Maintenance (ICSM). (1995) 124–133
5. Mehta, A., Heineman, G.T.: Evolving legacy system features into fine-grained components. In: Proc. 24th Int. Conf. on Softw. Engg. (2002) 417–427
6. Kim, H.S., Chae, H.S., Kim, C.H.: A use-case based component identification approach for migrating legacy code into distributed environment. In: LNCS. Volume 3280., Springer Verlag (2004) 897–906
7. van Deursen, A., Kuipers, T.: Identifying objects using cluster and concept analysis. In: Proc. 21st Intl. Conf. on Softw. Engg., IEEE Computer Society Press (1999) 246–255
8. Li, S., Tahvildari, L.: A service-oriented componentization framework for java software systems. Proc. Working Conf. on Reverse Engg. (2006) 115–124
9. Chen, F., Li, S., Chu, W.C.C.: Feature analysis for service-oriented reengineering. In: Proc. 12th Asia-Pacific Software Engineering Conference (APSEC'05), IEEE Computer Society (2005) 201–208
10. Hess, H.M.: Aligning technology and business: Applying patterns for legacy transformation. IBM Systems Journal **44**(1) (2005) 25–46
11. Allen, J.R.: Dependence Analysis for Subscripted Variables And Its Application to Program Transformation. PhD thesis, Rice University, Houston, TX, USA (1983)
12. Field, J., Ramalingam, G., Tip, F.: Parametric program slicing. In: POPL '95: Proc. 22nd Symp. on Principles of Progr. Langs. (1995) 379–392
13. Fox, C., Danicic, S., Harman, M., Hierons, R.M.: Consit: a fully automated conditioned program slicer. Softw. Pract. Exper. **34**(1) (2004) 15–46
14. Cohen, Y., Feldman, Y.A.: Automatic high-quality reengineering of database programs by abstraction, transformation and reimplementation. ACM Trans. Softw. Eng. Methodol. **12**(3) (2003) 285–316
15. Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B., Wolfe, M.: Dependence graphs and compiler optimizations. In: Proc. 8th Symp. on Principles of Progr. Langs. (POPL). (1981) 207–218
16. Ball, T., Horwitz, S.: Slicing programs with arbitrary control-flow. In: Proc. 1st Int. Workshop on Automated and Algorithmic Debugging, Springer-Verlag (1993) 206–222
17. Komondoor, R., Ramalingam, G.: Recovering data models via guarded dependences. In: Proc. Working Conf. on Reverse Engg. (2007)
18. Lakhotia, A., Deprez, J.: Restructuring programs by tucking statements into functions. Inf. and Softw. Technology **40**(11-12) (November 1998) 677–689
19. Komondoor, R.: Automated duplicated-code detection and procedure extraction. PhD thesis, University of Wisconsin-Madison, USA (2003)
20. Stevens, W., Myers, G., Constantine, L.: Structured design. IBM Systems Jrnl. **13**(2) (1974) 115–139

A Discussion on Step 5 – procedure extraction

Identifying candidate code fragments for extraction into services and presenting them to the user (as illustrated in Figure 1) is the primary contribution of our approach. Hereafter, and after potential modification of these fragments by

the user, our approach provides for automated extraction of the fragments into separate procedures. For this we leverage our previously reported algorithm for procedure extraction [2]. Recall that for any seed S, Step 4 of the algorithm computes a bounding e-hammock $\mathcal{H}(S)$ that contains the candidate fragment to be extracted. Step 5 invokes the procedure extraction algorithm of [2] on the this bounding e-hammock, after the "marking" the following nodes within $\mathcal{H}(S)$ for extraction: the nodes that are in $f_{init}(S)$, plus all conditionals and loop headers in $\mathcal{H}(S)$ that control nodes in $f_{init}(S)$ (this is required for semantics preservation). The extraction algorithm splits $\mathcal{H}(S)$ into a sequence consisting of an e-hammock (potentially empty), a hammock, and then another e-hammock (potentially empty); the first e-hammock contains unmarked nodes in $\mathcal{H}(S)$ that could be moved out before, the last e-hammock contains unmarked nodes that could be out after, while the hammock (in the middle) contains all marked nodes (plus unmarked nodes in $\mathcal{H}(S)$ that could not be moved out of the way). The hammock is then ready to be pulled out into a separate procedure, and replaced by a call to this procedure.

The extraction algorithm is quite general; for instance, it extracts contiguous fragments (e.g., (2)), non-contiguous fragments (e.g., (3)), and even fragments that contain jumps whose targets are not the fall-through exit of the fragment (not exhibited in our running example). The algorithm does *not* classify the variables in the extracted fragment as parameters and local variables; this is relatively straightforward, e.g., as discussed in [18].

Figure 5 shows the result of extracting the candidate fragments in our running example. The left column and the top of the right column show the extracted procedures. For brevity, we omit the declarations of the variables, as well as information on whether they are local variables or parameters (e.g., `ord-item-rec` is a local variable in service (2), with declaration just as in Figure 1). The approach *does not* automatically give names to the services; these have been provided in the figure manually for clarity[5]. Code refactoring can be performed across the extracted services to capture commonalities, and this is orthogonal to our approach.

Another capability our algorithm inherits from the extraction algorithm is rewriting the batch program to use the extracted procedures. This is shown in the right hand column of the figure. This can be useful if the batch program is to be retained along with the new online system (say, as a transitional measure), and there is a desire to minimize code duplication between the old and new systems. Note that the rewritten batch program contains essentially the calls to the extracted services, plus the statements in Figure 1 that are *not* part of any candidate service (including line 37, which was moved out of the way of fragment (3))[6]. Note that line 38, which was identified as a separate service, in

[5] Similarly, we do not address generation of WSDL descriptions for the services, generation of plumbing code to implement parameter passing, etc. These issues are orthogonal to our main contribution.

[6] Certain conditionals (see line (36)) that are included in a service are also duplicated in the rewritten batch for correctness of the transformation.

the end gets included in service (3) because it could not be moved out of the way by the extraction algorithm.

For further illustration, we consider service fragment (3) in Figure 1. As mentioned in Step 4 in Section 3.2, the bounding e-hammock $\mathcal{H}(S)$ for this fragment is lines 26 through 43. We apply the extraction algorithm on this e-hammock, with all lines being marked except lines 37 and 38 (which are not part of the fragment (3)). The extraction algorithm creates two e-hammocks, as shown in Figure 5: (a) an hammock containing lines 26 through 43, including line 38, which it could not move away without guaranteeing semantics preservation, but excluding line 37, and (b) an e-hammock containing lines 36, 37, and 43. Hammock (a) is extracted as procedure `FullOrder`, and replaced by a call to this procedure. E-hammock (b) is left behind at the point following the procedure call.

```
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 eof-flag PIC x(6).
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 'no' TO eof-flag.
/7/ MOVE 0 TO num-fulfilled.
/8/  PERFORM UNTIL eof-flag = 'eof'
/9/    READ in-file AT END
/10/     MOVE 'eof' TO eof-flag
/11/     GO TO LOOP-END.
/12/   MOVE in-rec TO order-rec.
/13/   IF ord-type = 'regord'
/14/     ADD 1 to num-fulfilled
/15/     WRITE order-rec TO out-file              (1)
/16/     i = 1
/16a/    WHILE i <= ord-num-items
/17/       READ in-file
/18/       MOVE in-rec TO ord-item-rec            (2)
/19/       READ item-table KEY IS ord-it-id       (2)
/20/       IF item-avbl-count < ord-it-count      (2)
/21/         MOVE item-avbl-count TO ord-it-count (2)
/22/       ENDIF                                  (2)
/23/       WRITE ord-item-rec TO out-file         (2)
/24/       i = i + 1
/24a/    END-WHILE
/25/   ELSE IF ord-type = 'fullord'
/26/     MOVE 'ok' TO full-all-avbl               (3)
/27/     i = 1                                    (3)
/28/     WHILE i <= ord-num-items                 (3)
/29/       READ in-file                           (3)
/30/       MOVE in-rec TO ord-item-recs(i)        (3)
/31/       READ item-table KEY IS ord-its-id      (3)
/32/       IF item-avbl-count < ord-its-count(i)  (3)
/33/         MOVE 'notOk' TO full-all-avbl        (3)
/33a/      ENDIF                                  (3)
/34/       i = i + 1                              (3)
/35/     END-WHILE                                (3)
/36/     IF full-all-avbl = 'ok'                  (3)
/37/       ADD 1 to num-fulfilled
/38/       WRITE order-rec to out-file            (4)
/39/       i = 1                                  (3)
/40/       WHILE i <= ord-num-items               (3)
/41/         WRITE ord-item-recs (i) TO out-file  (3)
/42/         i = i + 1                            (3)
/42a/      END-WHILE                              (3)
/43/     ENDIF                                    (3)
/44/   ENDIF
/45/ LOOP-END:
/46/ END-PERFORM
/47/ WRITE 'Num. orders fulfilled = ',
       num-fulfilled TO out-file.
```

Fig. 1. Example batch program

Step 1. Identify all "tag" conditionals. A "tag" conditional is an "if" conditional that determines the kind of an input transaction (e.g., regular order or full order).
Mark all `WRITE` statements "unprocessed".
While there remain unprocessed `WRITE` statements do
  Step 2. Identify a "seed" S, where S is a maximal group of unprocessed `WRITE` statements in a single procedure controlled by the same set of tag conditionals and within the same loops.
  Step 3. Identify an initial candidate fragment $f_{init}(S)$ by applying a variant of backward slicing from all statements in S and taking a union of the results.
  Step 4. Prune the initial candidate fragment $f_{init}(S)$ to make it have certain desirable properties.
  Step 5. Extract the pruned candidate fragment into a separate procedure, and replace the fragment in its original location by a call to the new procedure. Mark all statements in S as "processed".

Fig. 2. Outline of algorithm

$f_{init}(S) = S$
repeat
  while $\exists m, n \mid (m \xrightarrow{f} n) \wedge (n \in f_{init}(S)) \wedge (m \notin f_{init}(S)) \wedge$ (m not an input statement on an input file) **do** {backward dataflow slicing}
    Add m to $f_{init}(S)$. Mark $m \xrightarrow{f} n$ visited.
  end while
  for all $l \in \mathcal{LH}$ **do** {Identify loops with non-independent iterations}
    if $\exists cycle = (n_1 \xrightarrow{f} n_2 \xrightarrow{f} n_k \xrightarrow{f} \ldots \xrightarrow{f} (n_{k+1} = n_1))$ **then**
      if $\forall 1 \leq i \leq k \mid (n_i \in loop(l) \wedge (n_i \xrightarrow{f} n_{i+1}$ is marked visited)) **then**
        if $\neg \exists l_1 \mid (l_1$ is a loop header nested inside $l) \wedge (cycle$ is contained in $loop(l_1))$ **then**
          Add all statements and conditionals in $loop(l)$ to $f_{init}(S)$.
        end if
      end if
    end if
  end for
until the set $f_{init}(S)$ does not grow any more

Fig. 3. Variant of backward slicing for identifying initial candidate fragment

$f_{core}(S) = \{n \mid (n \in S) \vee (\exists l \in \mathcal{LH} \mid (n \in loop(l)) \wedge (loop(l) \subseteq f_{init}(S)) \wedge (S \subseteq loop(l)))\}$

$\mathcal{H}(S) = H$, where H is the most deeply nested, shortest sequence of (simple or compound) statements such that

- no node in H other than its entry node is the target of any jump outside H
- H contains no jump j whose target is outside H and is postdominated by the entry node of H. This restriction exists to ensure correctness in a certain corner case, the details of which we omit (see [19]).
- H contains all nodes in $f_{core}(S)$
- H contains no tag conditional p such that $p \notin f_{core}(S) \wedge (\exists n \in f_{core}(S) \mid p \xrightarrow{c} n)$, and no loop header l such that $l \notin f_{core}(S) \wedge (\exists n \in f_{core}(S) \mid n \in loop(l))$.
- any other CFG subgraph H' that satisfies all above properties and contains H contains no node n such that $n \in f_{init}(S) \wedge n \notin H$. (That is, intuitively, H is a *tight* bounding e-hammock.)

Fig. 4. Identifying an e-hammock $\mathcal{H}(S)$ that bounds a subset of $f_{init}(S)$

```
OrderHeader1(order-rec, out-file)
/15/ WRITE order-rec TO out-file.              (1)

OrderHeader2(order-rec, out-file)
/38/ WRITE order-rec TO out-file.              (4)

RegOrderItem(in-rec, out-file)
/18/  MOVE in-rec TO ord-item-rec              (2)
/19/  READ item-table KEY IS ord-it-id         (2)
/20/  IF item-avbl-count < ord-it-count        (2)
/21/    MOVE item-avbl-count TO ord-it-count   (2)
/22/  ENDIF                                    (2)
/23/  WRITE ord-item-rec TO out-file           (2)

FullOrder(order-rec, out-file, full-all-avbl)
/26/  MOVE 'ok' TO full-all-avbl               (3)
/27/  PERFORM VARYING i FROM 1 BY 1            (3)
/28/  UNTIL i > ord-num-items                  (3)
/29/    READ in-file                           (3)
/30/    MOVE in-rec TO ord-item-recs(i)        (3)
/31/    READ item-table KEY IS ord-it-ids      (3)
/32/    IF item-avbl-count < ord-it-counts(i)  (3)
/33/      MOVE 'notOk' TO full-all-avbl        (3)
/34/    ENDIF                                  (3)
/35/  END-PERFORM                              (3)
/36/  IF full-all-avbl = 'ok'                  (3)
/38/    CALL OrderHeader2(in-rec, out-file).   (4)
/39/    PERFORM VARYING i FROM 1 BY 1          (3)
/40/    UNTIL i > ord-num-items                (3)
/41/      WRITE ord-item-recs (i) TO out-file  (3)
/42/    END-PERFORM                            (3)
/43/  ENDIF                                    (3)
```

```
BatchTrailer(num-fulfilled, out-file)

Rewritten batch program.
/1/  OPEN INPUT in-file
           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 'no' TO eof-flag.
/7/  MOVE 0 TO num-fulfilled.
/8/  PERFORM UNTIL eof-flag = 'eof'
/9/   READ in-file AT END
/10/    MOVE 'eof' TO eof-flag
/11/    GO TO LOOP-END.
/12/  MOVE in-rec TO order-rec.
/13/  IF ord-type = 'regord'
/14/    ADD 1 to num-fulfilled
/15/    CALL OrderHeader1(in-rec, out-file). (1)
/16/    PERFORM VARYING i FROM 1 BY 1
        UNTIL i > ord-num-items
/17/      READ in-file
/18/      CALL RegOrderItem(in-rec,out-file).(2)
/24/    END-PERFORM
/25/  ELSE IF ord-type = 'fullord'
/26/    CALL FullOrder(in-rec, out-file,
              full-all-avbl).                  (3)
/36/    IF full-all-avbl = 'ok'
/37/      ADD 1 to num-fulfilled
/43/    ENDIF
/44/  ENDIF
/45/ LOOP-END:
/46/ END-PERFORM
/47/ WRITE 'Num. orders fulfilled = ',
       num-fulfilled TO out-file.
```

Fig. 5. Services extracted, and rewritten batch program, for example in Figure 1