

A Case Study in Matching Service Descriptions to Implementations in an Existing System

Hari S. Gupta*, Deepak D’Souza*, Raghavan Komondoor* and Girish M. Rama†

*Indian Institute of Science, Bangalore, India

Email: hari.csa.iisc@gmail.com, {deepakd, raghavan}@csa.iisc.ernet.in

†Infosys Technologies Ltd., India

Email: Girish_Rama@infosys.com

Abstract—A number of companies are trying to migrate large monolithic software systems to Service Oriented Architectures. A common approach to do this is to first identify and describe desired services (i.e., create a model), and then to locate portions of code within the existing system that implement the described services. In this paper we describe a detailed case study we undertook to match a model to an open-source business application. We describe the systematic methodology we used, the results of the exercise, as well as several observations that throw light on the nature of this problem. We also suggest and validate heuristics that are likely to be useful in partially automating the process of matching service descriptions to implementations.

I. INTRODUCTION

A large number of organizations are saddled with monolithic applications that combine too many different independent functionalities. Many of these organizations are in the process of migrating these applications to a Service Oriented Architecture (SOA) [1], [2], with the goal of improving the maintainability of the applications, the reuse-ability of individual functionalities within the applications, and the potential for integration with other applications. A common approach to this is to start with an analysis of the business domain, identify the important business processes, and use these to create a model of the required services [3]. Once the SOA domain model is finalized, it is typically realized by reusing matching portions of the existing monolithic implementation. These matching portions are either wrapped into services, or used as a basis for writing new code. Thus the ability to reuse the functionality already implemented in the existing system is widely agreed to be the key to successful migration to SOA.

However, given an abstract service description, locating parts of the source code in the existing system that implements that service is not easy, and has been a challenge for the SOA research community [4]. There are several reasons for this. In large monolithic systems, generally the source code runs into millions of lines of code making it hard to understand and search. Moreover, knowledgeable developers might have left exacerbating the problem of locating relevant portions of code. Systems documentation is sparse or in some cases non-existent. There is often a lot of code serving utility purposes (i.e., “plumbing code”), that gets in the way of matching core business services to their implementations. The most challenging aspect of this problem, however, is that the terminology used at the level of business process is likely to

be different from the one used at the code level to name files, variables and functions.

For of all these reasons, manually identifying parts of code that implement a given service is infeasible for large systems with thousands of files. At the same time, almost all practitioners we talk to agree that a completely automated approach is unlikely to yield good results due to the complexity of the problem. Therefore, we believe, a semi-automated approach where a knowledgeable developer follows a clear methodology with tool support is the way forward. Even here, it is not clear what methodology or heuristics a developer ought to follow while attempting to identify whether an abstract service has been implemented in the source code, and if so how. What would be very helpful is a detailed, real-life case study of the problem of matching a model of services to an implementation. Such a study would identify the challenges in this problem, suggest a road-map of specific technical problems to solve in order to arrive at solutions, and come up with some initial results towards a solution. Unfortunately, to the best of our knowledge, there are no case studies of this nature reported in the literature.

The goal of this paper is to address these issues. We carry out a detailed case-study to identify (a) the feasibility of and challenges in this problem, (b) the characteristics of a good match between a model and an implementation, and (c) features in code and heuristics that are most suited to (partially) automate a solution to this problem. In our case study, we began with a structured list of service descriptions in the ERP domain provided to us by practitioners, and attempted to match these in the source code of an open-source Java-based ERP system called JAllInOne. We first manually located portions of code in JAllInOne that implemented the given services in the most precise manner possible. We then designed some semi-automated heuristics that we hypothesized would help partially automate a solution to the problem, and evaluated them against our (ideal) manual matching. These set of heuristics, implemented as tools, can assist a developer in matching service implementations in abstract service descriptions to the existing code.

The contributions of this work are:

- A detailed manual methodology for mapping a model to an implementation. We believe ours is the first approach to match a real domain model to a real application to the

fullest extent possible. Another novelty is the way we use code features to *find* matches for services, and then use the GUI to *validate* our matches.

- We describe our experiences during the matching, show representative results, highlight challenges, say what works and what does not work, and justify the strengths of our proposed methodology.
- We make several observations about the structure of the application we analyze (which is typical of monolithic business applications in many ways), that are likely to be useful to researchers and practitioners working in this area. Due to lack of space, we present these observations in an accompanying technical report [5].
- We present a set of automated heuristics for model to code matching, that could be useful in partially automating the matching problem. We validate them against our earlier mentioned manual study, and identify which ones among them work well and which ones do not work so well.

The rest of the thesis is structured as follows. We describe the real-life model as well as the application that we use for our case study in Sec. II, goals, challenges and overview of the case study in Sec. III, and step by step manual methodology used in the case study in detail in Sec. IV. We propose and evaluate certain heuristics for the matching problem in Sec. V. We survey related work in Sec. VI and mention directions for future work in Sec. VII.

II. DESCRIPTION OF MODEL AND APPLICATION

A. The model

The key artifacts used in our case study were a *model* and an application. The model was created independently by domain experts in a major global software services company, and is an English language description of a representative set of services required in the ERP (Enterprise Resource Planning) domain. A service is a user-recognizable high-level functionality. A subset of the model is shown in Fig. 1. In the model each service has a name and a description; often the description is very brief, and does not contain much information beyond what is implied by the name. As can be seen, services are grouped into *collections*, and collections into *groupings* (there are other collections within the Sales Execution grouping that are not shown in the figure). A service collection refers to a set of services acting on a common *entity*, and differing only in their *action* on the entity. For instance, Fig. 1 shows the service collection Manage Sales Order In (referred to hereafter as “Sales Order;”) containing several services that pertain to different actions on sales orders. It is possible for multiple collections to be based on the same entity. A grouping is a set of service collections acting on related entities. For instance, the sales-execution grouping shown in the figure contains other collections such Manage Customer Returns In and Ordering Out. Examples of other groupings in the model are Account Management, Demand Fulfillment, and Demand Planning. Fig. 2, column 2, shows the total number

- sales-execution [Grouping]

Manage Sales Order In [Collection of services]

[Items below are services]

- Change Sales Order
- Change Sales Order Item Request to and confirmation from the Sales Order Processing to change a Sales Order Item
- Change Sales Order Item Schedule Line A request to and confirmation from Sales Order Processing to change a schedule line of a sales order item
- Check Sales Order Creation Query-and-response operation that communicates with Sales Order Processing to establish whether a sales order can be created with given data
- Update Sales Order
- Check Sales Order Update
- Create Sales Order A request to and confirmation from Sales Order Processing to create a sales order.
- Read Sales Order A query to and response from Sales Order Processing to provide order data.
- Read Sales Order Item
- Read Sales Order Query to and response from Sales Order Processing to read a sales order.
- Find Sales Order Item Basic Data by Elements Find Sales Order Item by assignment to WBS element
- Cancel Sales Order
- Confirm Sales Order
- Find Sales Order Basic Data by Buyer and Basic Data Query to and response from Sales Order Processing to retrieve basic data about sales orders, restricted according to customer and basic data.

Fig. 1. A fragment our domain model

	Model	UI	Model ↔ UI match	Model ↔ code match
Groupings	10	11	4	-
Collections	48	94	9	9
Services	191	246	34	36

Fig. 2. Statistics about the manual matching

of groupings, collections, and individual services in the model we use.

As stated in the Introduction, a domain model such as this one plays a very important role in any exercise in identifying service implementations in an application. The model helps fix both the exact nature of services sought, as well as their granularity. Approaches that do not use a domain model are likely to find services that are too fine grained or too coarse grained for the requirement in hand. Also, matching against a model makes it trivial to assign meaningful names to identified service implementations, which is very useful for the usability of the inferred services.

The model used in this study is the subset of the full model developed by domain experts. Since the full model is a proprietary one, getting even a subset of it for our study was very difficult. For all these reasons our part of the model does not capture most of the functional components of the business domain e.g., services corresponding to customer relationship management, employees, administration, etc..

B. The application

For our study we use the open-source application JAIII-nOne [6], which is an ERP application designed for medium and small-scale companies. It is developed using OpenSwing framework [7]. The reasons why we chose JAIII-nOne apart from the fact that it was open-source, are that it is in the same domain as the model (i.e. ERP), is reasonably well-modularized, making the manual study somewhat more tractable, and is written completely in a single language –

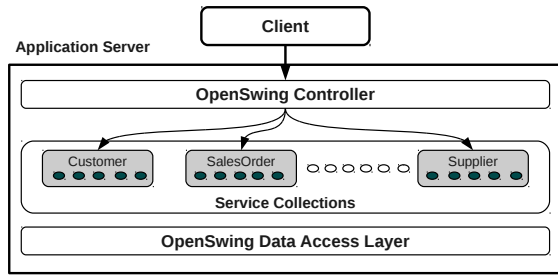


Fig. 3. JAllInOne architecture

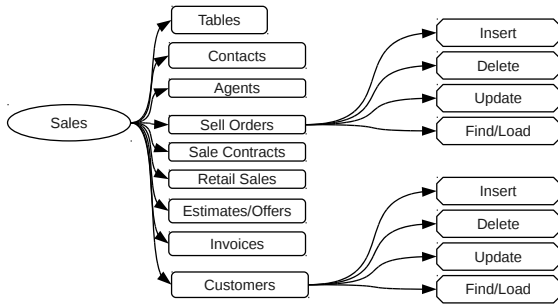


Fig. 4. Abstract representation of a subset of the JAllInOne UI

Java – making it a good test-candidate to implement analysis techniques to partially automate the problem of service mining. JAllInOne has 1089 files (classes), contained in 258 directories and subdirectories, with 223,241 lines of code.

The architecture of JAllInOne is depicted in Fig. 3. It consists of a UI client, a “server” which contains the files that provide actual functionality, and a “controller” which receives commands from the client and invokes appropriate files in the server. Though our study does involve running the application and invoking its UI, while analyzing the code we restrict our attention to the server. Hence, when we refer to classes or files in the application we mean the java classes in the server.

The UI of JAllInOne is interesting in that it can be abstracted as a three tiered structure in which tiers correspond to groupings, service collections, and actions (similar to the three tiers in the model). We consider only those buttons (or menu items) on the application UI as actions which invoke one or more server side functions. We depict a part of the UI abstraction in Fig. 4. From left to right the nodes depict a grouping (i.e., Sales), some collections under this grouping, and services under two of the collections (Sell Orders and Customers). Abstraction of UI helps in matching the model to UI at an abstract conceptual level. Once we have abstract representation of the UI, we can simply invoke the UI action corresponding to a model service to collect its execution trace, which we use in validation phase. The third column of Fig. 2 shows statistics about the UI.

Hereafter whenever we refer to the UI, we mean the UI abstraction, as shown in Fig. 4. Please refer to our accompanying technical report [5] for details on how we produced this abstraction.

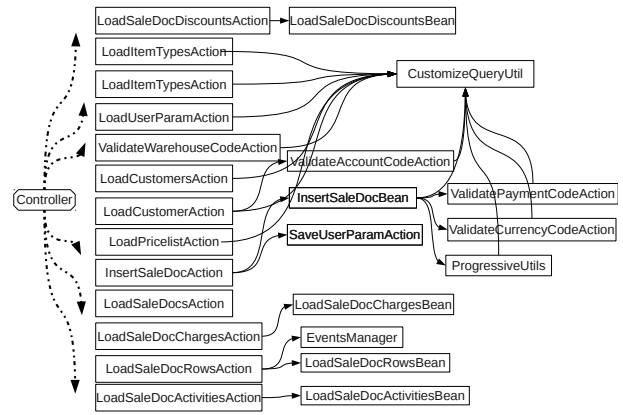


Fig. 5. Execution trace of Insert Sell Order

III. GOALS OF THE STUDY, AND CHALLENGES FACED

The goals of our manual case study were to match as many services in the model as possible to the application. Based on the observation that every service is an action acting upon a business entity corresponding to a collection in the model, we proceeded in two steps. We first matched collections in the model to the application. For each matching collection we identified a set of files in the code, which we call the *collection implementation*, that implements the functionality of the collection. We point out here that we would like the collection implementations to correspond to the “core modules” that implement the functionality of the collection, and hence we require the collection implementations to be disjoint. In the second step we matched the services in the matching collection to subsets of files contained in the collection implementation.

Another goal was to do this matching as precisely as possible, so we could identify (a) the feasibility of and challenges in this problem, (b) the characteristics of good matches, and (c) features in code and heuristics that were most suited to (partially) automate a solution to this problem.

Our first thought was to do the matching by (a) matching the services in the model to the actions in the UI (described in Section IV), and (b) executing each action in the UI, recording the files reached in the execution trace, and assigning these files to the service that was executed. We realized however, that there were various difficulties with this approach that made it infeasible, as listed below.

- In many cases the UI does not have sufficient features to allow us to guess with confidence whether a certain UI element matches an element in the model. For instance there is a collection Vehicle Movement in the model. The same label does not appear anywhere in the UI, but the UI does have an element Goods Movement. It is hard to ascertain whether the two ought to match or not. In other words, we do not have very high confidence in our model-to-UI matching. In contrast, in the code there are a number of other features (e.g. names of identifiers, comments, names of files and directories) that increase

the confidence of our matching.

- Although we earlier mentioned that the UI is organized, like the model, in terms of groupings, collections, and actions, in fact many of the leaf elements of the UI (i.e., actions) are not simple services, but compositions of services (i.e., mini business processes). For instance, when we execute the action Insert under “Sell Orders” in Fig. 4, which matches to the service “Create Sales Order” in the model (see Fig. 1), the UI makes us select a customer from a list of customers, an item from a list of items, etc., in order to populate the sales order. The fact that each UI action is actually a mini process, is confirmed by the trace of files that are reached during this execution, as shown in Fig. 5. The files in the first column (after the Controller) get invoked directly by the Controller, whereas the other files are invoked transitively (through a chain of invocations). In our understanding only the files InsertSaleDocAction (called by the Controller) and InsertSaleDocBean (called by InsertSaleDocAction) constitute the implementation of Create Sales Order.
- The UI does not directly expose to the user all the services that exist both in the model and in the code. For instance, Confirm Sales Order and Validate Sales Order are services in the model that have a matching implementation in the code, but do not match any action in the UI. The files corresponding to these services are invoked implicitly as part of other composite actions in the UI; therefore, it would be difficult to match these files to their respective matching services using just the information in the trace.
- Several of the files in Fig. 5 are utility files, with no business logic (e.g. EventsManager). These ought not be included in the implementation of any collection. It would be hard to identify and separate out such utility files using the traces only.
- Not all applications have a UI; for example batch processing systems which are used in the back-end of several existing systems. We would like our case study methodology and results to also extend to applications without UIs. Hence we choose to use the UI, when available, for validation of a model-to-code match, and to not require a UI to be present to do the matching.

For these reasons we decided to follow the approach of independently matching the model to the code (as described in Section IV). After matching the model to the code, we used execution traces to validate this matching (Section IV). This entire process is summarized in Fig. 6.

The model to code matching presented its own set of issues, namely (a) large size of the application, (b) ambiguities in deciding the boundaries between the implementations of different collections/services, (c) the use of terminology in the identifiers and comments in the application that is different from that in the model, e.g., actions (insert vs. create, load vs. “read” or “find”, etc.), terms present in the description

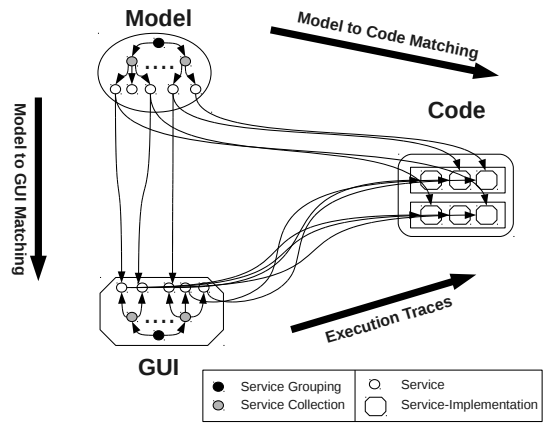


Fig. 6. Three step approach to matching model with application.

of services, some collection names (e.g., item vs. material), and (d) the presence of a large number of non-business-logic related (utility) files. We describe in Section IV how we overcame these difficulties. One difficulty we did *not* face was interleaving of code corresponding to different services or collections within a single file. JAllInOne is well-modularized in this respect, whereas older applications in legacy languages are often not [8]. Since our focus was on understanding the issues and difficulties listed above, it helped us that our application did not present to us the additional difficulty of interleaving.

IV. STEPS IN MANUAL METHODOLOGY

Step 1: Matching model to code

The goal of this part of the case study is to match each collection (and within that, each service) in the model to its implementation within the code. The methodology we followed to match a collection C was as follows. Firstly, identify a set of *seed* files for the collection using features in the files that strongly associate it with the collection. Once we had seed files for collection C , we followed call-graph edges from these files, in both directions, looking to add neighboring files to the collection. We call the process “expansion”, and it terminates when no more files can be associated with collections. We verified at the end of the process to the best of our ability that the files associated with each collection indeed constitute the implementation of the collection.

We now present the details of the approach summarized above. Firstly, we enumerate certain properties of files that we use in the seed-finding and expansion steps.

- P_1 : Accesses database tables (T_C) pertinent to the collection C .
- P_2 : Accesses majority of the fields (attributes) of one or more tables in T_C .
- P_3 : Name of the file has some similarity with the collection name or a table name in T_C .
- P_4 : Contains comments indicating its relevance to the collection C .

- P_5 : Most of the callers and callees feature most of the above properties; i.e. the file is surrounded by other files that belong to the implementation of collection C .
- P_6 : The file is located in the directory where most of other files of the specific collection are located.

A set of rules R_S that a file should satisfy to be a seed file of the collection C , is given below.

- S_1 : The file exhibits property P_1 , and
- S_2 : The file conforms to the majority of the other five properties ($P_2 - P_6$), and
- S_3 : The file shows closer proximity (based on S_1 and S_2) to the given collection C than other collections.

Similarly in expansion phase, a file was added to the implementation of the collection C if either rules E_1 , E_2 and E_3 ; or rules E_2 and E_4 ; or all four rules; were satisfied:

- E_1 : Satisfies some of the properties among $P_1 - P_6$.
- E_2 : Has close proximity (i.e., called by or calling) to a seed-file or any other file already assigned to the collection implementation.
- E_3 : Shows closer proximity (based on E_1 and E_2) to the given collection than other collections.
- E_4 : Is not called by any of the files of any of the other collections.

As we observe in rules S_2 , S_3 , E_1 and E_3 , we often took subjective decisions using our intuition to include a file in a collection.

Once each collection was assigned a set of files, we partitioned this set of files among the services in the collection. This is often easy to do; the seed files of a service often contain in their name or in the names of identifiers within them the *action* word associated with the service (e.g. load, create, delete). Once the seed files of each service within a collection have been identified in this way, we expand from the seeds and hence partition the set of files associated with the collection among its services using a process similar to the one described above for associating files with collections.

The numbers corresponding to the matching process of this step are summarized in the last column in Fig. 2. The number of source files we identified for each collection is shown in Figure 11 (the black portions plus the white portions). The accompanying technical report [5] mentions the names of these files for all nine collections, a link to the source code of the exact version of JAllInOne we analyzed (0.9.21), and other details.

Note that most of the collections in the model did *not* have a match in UI or in application source code. This was due to following reasons: The domain model was meant for large companies, and had groupings such as Product Development, Supply Planning, and Demand Planning, which had no match in JAllInOne, which is meant for small enterprises. At the same time we were given access to only a subset of the model, in which were missing key groupings like CRM and Production, which are present in JAllInOne. This said, the matching exercise still gave us a lot of insight into the challenges involved, even though the actual number of matches

was small as a proportion of the total model/application.

Illustration of model to code matching: As an example, we depict in Fig. 7 the files we matched with the Sales Order collection in the model (see Fig. 1), along with the files which are in the same directory as the matching files, and the immediate callers and callees of all these files. Each node in Fig. 7 represents a file in the implementation, with the edges representing call edges (the files without any call-edges incident are the entry points). The dashed call-edges are ones whose other end is not depicted. Each dashed call-edge in fact represents one or more call-edges with the same direction. The gray boxes are the seed files and the double boxes are the files obtained by *expansion*. All other files depicted in the figure are ones that were not included in the Sales Order collection implementation. The following examples describe a few representative cases in seed finding and the expansion process.

We first determined that among the 119 database tables used in the application DOC01_SELLING and DOC02_SELLING_ITEMS are the tables most pertinent to the Sales Order collection (see Fig. 9 for a list of other matching collections, and their pertinent tables); i.e., these constitute the set $T_{SalesOrder}$. Based on this determination we identified the seed files of the collection satisfying the rules $S_1 - S_3$ (see gray boxes in Figure 7).

“CloseSaleDocAction” was the only ambiguous seed file for Sales Order collection. The file “CloseSaleDocAction” was strong in all features except P_5 . Based on the name and comment inside (“Action class used to close a sale document...”) we considered it a strong candidate for being a seed. It satisfied all the three rules ($S_1 - S_3$) for a seed. It however did not show very high confidence for S_3 . This shows that the decision of assigning a file to a collection in the case where all features are not satisfied is a highly human dependent decision.

We now discuss expansion, in which we look for the files satisfying the rules given for expansion (i.e., E_1 , E_2 and E_3 ; or E_2 and E_4 ; or all $E_1 - E_4$). In the majority of cases non-seed files could be unambiguously placed in the Sales Order collection based on the given rule set. Certain non-seed files such as “LoadSaleDocRowsAction” are unambiguously placed in the Sales Order collection, because they connect only to seed files in the collection. On the other hand, several of the non-seed files were assigned to this collection unambiguously even though they are connected to files outside the collection. This was due to the preponderance of evidence to assign them to Sales Order. For instance, most of the files in this category have the words Sale and Doc in their name (which are the indicative features of this collection, based on the table names mentioned above). Most of the files also are in the same directory as all other files in this collection.

An interesting case we noticed was the file “InsertSaleSerialNumbersBean”. Although it did not satisfy most of the given features, it was called by only two files (“UpdateSaleDocRowAction” and “InsertSalesDocRowAction”), both of which had strong correspondence with the Sales Order

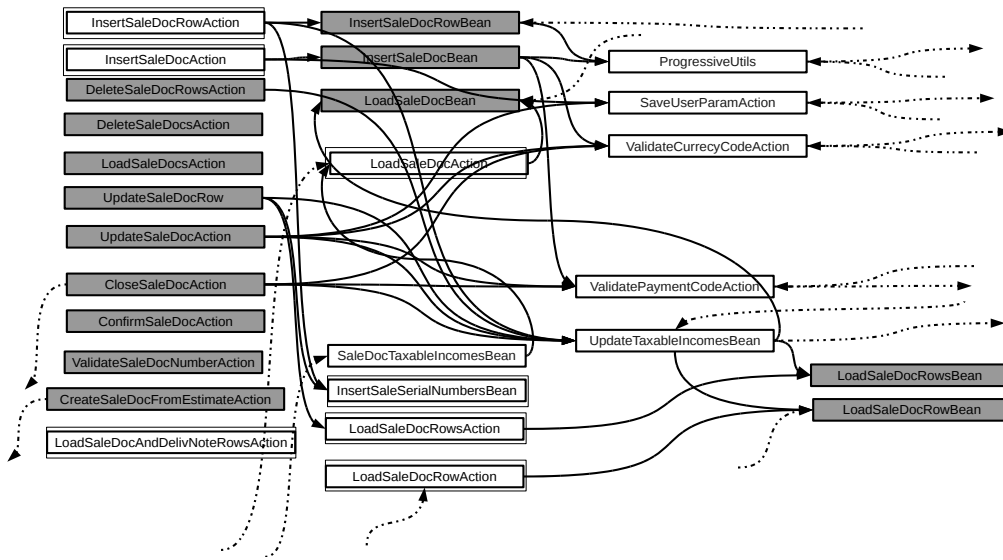


Fig. 7. Call graph for files of sales-order collection and other files in surrounding (callers, callees and files in the same directory). Gray boxes, double boxes, and plain boxes represent the seed files, files added in the expansion process, and the files outside the sales-order collection implementation, respectively.

collection. Therefore this file also was added to the Sales Order implementation.

For most of the remaining files we could easily decide that they were not pertinent to the Sales Order collection, by using the set of rules given for expansion. These files are located in the same directory as the seed files but do not access any of the tables pertinent to the Sales Order collection. The comments inside these files also do not give any confidence towards their relevance to the collection. We consider these files parts of other collections that may be present in the application. In brief these files do not satisfy the rules for being a seed file or an expanding file. Therefore we pruned these files unambiguously.

Some files like “UpdateTaxableIncomeBean” were ambiguous. Although the name of the file gave no indication that it ought to belong to this collection, upon closer perusal, we determined that the file (a) accessed the table DOC02_SELLING_ITEMS, and (b) had the comment “Description: Help class used to update all taxable incomes for all items and activities ... for the specified sale document ...”. Overall this file showed good relevance to Sales Order collection. But this file was accessing tables other than Sales Order tables and was called by and calling several files outside the Sales Order implementation (i.e., showed good relevance to files outside the Sales Order implementation). Due to all these reasons we did not include this file in the implementation.

Step 2: Matching model to UI

The goal of matching the model to the UI is to validate the model-to-code matching at finer level. We use the abstraction of UI for matching the model to the UI, which is illustrated in Fig. 4.

We summarize the results of the model-to-UI matching process in Fig. 2, Column 4, as well as Fig. 8. Four group

Model	UI
Purchasing and Sourcing (A): Supply management (2)	Purchases (A): New invoice from Delivery notes (8), Buying orders (3), Suppliers (2)
Account Management (B): Price management (4), Customer (5)	Accounting (B):
Sales Execution (C): Sales order (6)	Sales (C): Sell orders (6), Customers (5), Sale price list (4)
Warehousing and Storage (D):	Warehouse (D): Out delivery notes (9)
Supply Planning: Material (1)	Table: Items (1)
Procure to pay: Purchase order (3)	Production: Production orders (7)
Demand Fulfillment: Production order (7), Inbound delivery notes (8), Outbound delivery notes (9)	

Fig. 8. Summary of model to UI matching. Each entry is of the form **Group: collections**. Groups/collections with the same label match.

names in the model matched four group-names in the UI; these are shown in the first four rows in Figure 8, with labels A, B, C, and D, respectively. There were 9 pairs of matching collections between the model and UI. These are labeled 1 through 9 in Fig. 8 (each pair of matching collections have the same label in the figure). For brevity, we show only the matching groups and collections in the figure. Note that the matching groups/collections do not have identical names; we had to use our intuition and guess that they match.

Within the 9 matching collections there were 46 (resp., approximately 36) individual services in the model (resp., UI). Of these, 34 pairs of services matched each other.

Note that the model to UI matching was many-to-many; sometimes multiple model services matched a single UI element, and sometimes multiple UI elements matched a model service. However, most of matches were one-to-one.

Step 3: Validation of match using execution traces

As discussed in Section III, and depicted in Fig. 6, the goal of this step is to use the model-to-UI matching we obtained

(see Step 2 above) to validate the model-to-code matching (Step 1 above).

For any given collection C , let G be the set of actions in the UI that have matched the services inside C (by Step 2). We execute the actions in G and save the resulting set of execution traces E . We say that the validation for collection C has passed if all *entry points* in the implementation of C (as identified in Step 1 above) are reached in the set of traces E , where an entry point is defined as a file which is not called by any other server file. Intuitively, the validation determines if the entry points in the collection’s implementation have been identified *precisely*, in the sense that each one identified is indeed an entry point (is reachable during execution from some UI action that matches some service in the collection). We did not validate the precision of the non-entry-point files that we identified, nor we validate *recall* (in the sense that we are not sure if all files that ought to be entry points of a collection are indeed assigned to the collection). These would have required greater manual effort, in terms of exhaustively exercising the usage-scenarios afforded by the UI.

The results of the validation was such that of the 9 collections in the model that have matches (see Fig. 8), 7 of them passed the validation, except Sales Order and Purchase Order. The reasons for this are interesting to note. The service Confirm Sales Order in the Sales Order collection (see the second from last service in Fig. 1) matches a set of files in the code, but did not show up in any of the traces. Upon investigation we found that this service has no matching UI action (see the discrepancy in the number of services in the model that matched with the UI, versus those that matched with the code, in the last row in Fig. 2). Therefore, the entry point corresponding to the Confirm Sales Order service (see file ConfirmSaleDocAction in the leftmost column of Fig. 7) was never reached in any execution trace. The interesting thing to note here is that careful investigation of validation failures can either reflect the incompleteness of the UI with respect to the model (as in this case), or rectify problems with the model-to-code matching or in the model-to UI matching. In case of Purchase Order collection also validation fails due to the same reason.

V. HEURISTICS

In this section we first propose some filters. Each of these filters takes a set of source files as input and outputs a subset of the given set of source files corresponding to each collection accessed. Later in Section V-B, we present a basic semi-automated approach that uses these filters and matches the model to the code. We also evaluate the utility of the filters and the semi-automated approach by running them on the application under study, and comparing the output with the actual numbers obtained in our manual study in Section IV.

A. Filters

Each filter takes some parameters (for example a set of core tables, or a threshold value for keyword matching), and returns the subset of the source files in the application that

satisfy the parameters. Fig. 9 shows some of the manually identified features of the collections, that we will make use of as parameters to the filters described below. In the first column of the table, all the matching collection names are listed. The second column of the table shows the abbreviated collection names we use in the graphs later in this section.

The set F_C of source files returned by a filter F is meant to be an approximation of the set S_C of files that actually belong to a collection C (as identified manually in Section IV). For each filter F and collection C we record the number of “hits” (i.e., $|F_C \cap S_C|$), the number of “False Positives” ($|F_C - S_C|$), and the number of “False Negatives” ($|S_C - F_C|$). The number of false positives give us an idea of the “precision” of the filter, while the number of false negatives gives us an idea of the “recall” of the filter.

It should be noted that the precision and recall of a single filter should not be taken as the final precision and recall of the overall approach, since each filter outputs only an approximation of the actual implementation of a collection. We later combine all the filters, with some human intervention, to obtain a good semi-automated matching approach (see Section V-B). The filters presented here show how various observations can be exploited to match desired files and filter out most of the irrelevant files. The first two filters below (TA and TNA) are useful in identifying the “seed” files of a collection’s implementation.

1) *Tables Accessed (TA) Filter*: In our model each collection C correspond to a business entity. Therefore in its implementation, the database tables corresponding to this entity are likely to be accessed. This motivated us to design this filter which takes as parameter a set of core tables T_C (as shown in second column of Fig. 9), and returns the set of all source files that access a table in T_C . For example, for the service collection Sales Order, TA filter returns all the source files that access the table “DOC01_SELLING”.

2) *Tables Not Accessed (TNA) Filter*: This filter is meant to improve the precision of the TA filter above. We observed that the TA filter reports false positives – source files that access one of the given tables T_C of a collection C , but are not part of the collection. An example of this is the CUST collection. Its core table is “SAL07_CUSTOMERS.” However when we run the TA filter with this table as input, it reports (among others) source files pertaining to the Sales Order collection, since some services in Sales Order (for example Create Sales Order) access the SAL07_CUSTOMERS table, to access information related to the customer placing the sales order. Such information about certain tables that are definitely not accessed by a service the number of false positives produced by the TA filter.

The filter TNA takes as parameter the set of tables TNA_C not accessed by the collection C and outputs all source files that do not access any of the tables in TNA_C . Thus, in the case of the CUST service collection above, we include “DOC01_SELLING” in TNA_{CUST} before running the filter.

The next two filters, described below, help mainly with the process of “expanding” the files around the seed files to

Collection Name (C)	Abbreviated Name	Tables Accessed (T_C)	Tables Not-Accessed (TNA_C)	Related Words (RW_C)
Customer	CUST	SAL07_CUSTOMERS	DOC01_SELLING	name, address, ...
Inbound Delivery	IDN	DOC09_IN_DELIVERY...	—	delivery, item, ...
Material (Items)	ITM	ITM01_ITEMS, ...	DOC01_SELLING, ...	product, item, ...
Outbound Delivery	ODN	DOC10_OUT_...	—	warehouse, item, ...
ProductionOrder	PDO	DOC22_PRODUCTION...	—	product, order, ...
Purchase Order	PO	DOC06_PURCHASE, ...	—	purchase, supplier, ...
Sales Order (Sell Order)	SO	DOC01_SELLING, ...	—	sales, customer, ...
Sales PriceList	SPL	SAL01_PRICELISTS, ...	—	price, item, ...
Supplier	SUPP	PUR01_SUPPLIERS	DOC06_PURCHASE, ...	supplier, item

Fig. 9. Manually identified features for the collections

obtain the complete set of source files corresponding to a given collection. These filters assign a score to each remaining non-seed file, and output source files that have a score above a given threshold. The files in the output have high likelihood of being pertinent to the collection.

3) *Filename (FN) Filter*: This filter tries to exploit the fact that the names of source files corresponding to a collection C are often closely related to the canonical name used in the implementation to refer to the main entity operated upon by the collection. For example by examining the tables in T_{ITM} associated with the Material collection (see third row, third column in Figure 9), we inferred that “material” in the model is referred to as “item” in the implementation. Similarly, “sell order” is used in the implementation to refer to “Sales Order” in the model.

For each collection C , the FN filter works by first concatenating the words in the canonical implementation-name of this collection to obtain a string s , and then gives a score to each file, which is the length of the longest substring of s that occurs in the name of the file divided by the length of s itself. It then outputs the files whose score is above v , where v is a threshold parameter between 0 and 1. The canonical implementation-name is given as a parameter to the filter, as shown within parenthesis (wherever it differs from the name of the collection in the model) in Column 1 in Figure 9.

4) *Table Fields (TF) Filter*: The motivation for this filter is if a large percentage of the fields of a table $t \in T_C$ pertinent to a collection C are accessed in a file, but the table itself is not accessed in the file, then the file is pertinent to the collection. This happens, e.g., when a method in the file receives a row of data as a parameter from another file that accesses the table, and the method processes or prepares this row (by referring to the fields in the row).

This filter first gives a score to each file, which is the percentage of fields accessed of all the tables in T_C . The filter outputs source files whose score is more than a threshold parameter v .

We also considered using a filter based on terms related to the name of the collection (some example terms are given in the last column of Fig. 9). However we found that neither the precision nor the recall of this filter was not satisfactory. This was due to the fact that, (a) our set of related keywords

was small (due to our lack of domain expertise), and (b) the majority of files contain only one or two lines of meaningful comments. We provide the precision and recall statistics of all the above individual filters for each of the nine collections in the accompanying technical report [5].

B. A Semi-Automated Approach

We now suggest a semi-automated algorithm, using the filters defined above, to partially automate the procedure of matching the model of service descriptions to the source code. The algorithm is shown in Fig. 10.

Steps 2 to 7 of the algorithm are automated, whereas steps 1, and 8 to 10 are manual. The results we report in this section as shown in Fig. 11 were obtained by omitting steps 8 to 10. We used, 20%, 80% and 10% as the values of X_1 , Y_1 and X_2 respectively, and did not use the Step 7(c). We kept a threshold of a maximum number of files of 70 to tune the FN and TF filters in Step 5(a) and Step 5(b). We guessed this number (70) for tuning based on (a) the possible number of collections in the application, which we approximated based on database tables (119 tables in JAllInOne), and (b) the application size (1089 files in JAllInOne).

Note that the recall of the algorithm is 89%, 78%, 89%, 79%, 90%, and 100%, for IDN, ITM, ODN, PDO, SPL, and all other collections, respectively. We were satisfied with this performance. The precision, however, is less satisfactory, ranging from 100% (for the collection PDO) to 26% (for the collection ITM). While investigating this issue we found that most of the false positives (the names of which we provide in the accompanying technical report) were passing through FN filter. In case of collection ITM, the collection name in the implementation (i.e., “items”) is small; therefore, several irrelevant files also get approximately the same score as relevant files. For a similar reason collection SO gives a large number of false positives.

For collection ITM we find 2 false negatives. It is due to our selection of values for the parameters used in Step 3 and Step 7. These missed files (false negatives) have names that match poorly with the name of the collection, and access a table that has fewer fields than most other tables. Therefore these two files do not pass the test in Step 3. Two false negatives occurred for collection “Production Order” for similar reasons. These kinds of files can be found by designing a more sophisticated

- 1) User creates a table of features, as in Fig. 9.
- 2) For each collection C , apply the TA filter, followed by the TNA filter, to obtain a set of candidate seed files C_C .
- 3) For each file f in C_C , add f to an initial “seed” source files set M_C ,
 - a) if f ’s score due to FN filter is in the top $X_1\%$ of scores among all files in C_C .
 - b) else if f ’s score due to TF filter is in the top $X_1\%$ among all files in C_C .
 - c) if f ’s score due to each of the filters FN and TF is in the top $Y_1\%$ of scores among all files in C_C .
- 4) Create a temporary set C_T and add all files in M_C to it. (We use C_T to store candidate files for expansion.)
- 5) Do (perform expansion)
 - a) Apply the FN filter and tune the threshold value automatically so that only a small number of files are returned by the filter. Add a file among these to C_T , if it is an immediate neighbour in the call graph of a file already in C_T .
 - b) Apply the TF filter similarly.

While no new files are added.
- 6) Create a candidate expansion set, $C_E = C_T - M_C$.
- 7) For each file f in C_E , add f to M_C ,
 - a) if f ’s score due to FN filter is in the top $X_2\%$ of scores among all files in C_E .
 - b) else if f ’s score due to TF filter is in the top $X_2\%$ among all files in C_E .
 - c) if f ’s score due to each of the filters FN and TF is in the top $Y_2\%$ of scores among all files in C_E .
- 8) Manually analyze files in M_C and remove irrelevant files from it.
- 9) Expand M_C manually, by adding to it other files which are closely related to M_C in the call graph and are highly relevant to C .
- 10) If a file f is called by or calls files in M_C only, and is not contained in the set M_D of any other collection D , then add f to M_C .
- 11) Return M_C as the output set implementing the given collection C .

Fig. 10. A semi-automated algorithm to identify the implementation of a collection C , using TA, TNA, FN and TF filters.

TF filter that assigns separate scores to a file for each related table and then combines these scores in a meaningful way to assign a final score to the file. Some of other false negatives (e.g, one for “Indelivery Notes”, one for “Outdelivery Notes”, one for “Production Order”, etc.), we get because we did not use Step 10 while reporting the results.

To summarize, in this section we described and evaluated a preliminary semi-automated algorithm for matching collections to their implementations. With carefully chosen tuning

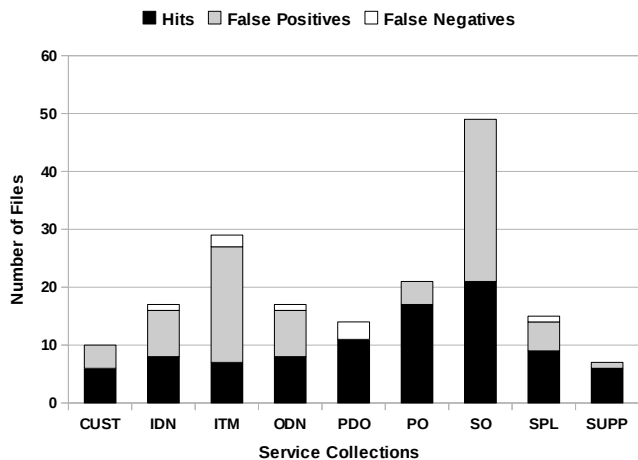


Fig. 11. Performance of semi-automatic approach

parameters we were able to achieve satisfactory performance with respect to recall. More work is required to investigate techniques to improve the precision of the approach.

VI. RELATED WORK

Mining services from monolithic applications is a well-researched area, with a number of approaches published in the literature. These approaches can be classified broadly into two categories, that we term as bottom-up service mining and top-down service mining.

In bottom-up service mining, the focus is on extracting reusable components from source code and wrapping them as services *without* a prior model of the required services. A number of approaches have been discussed in literature on identifying potential services from source code or similar problems. These include techniques based on software clustering [9], graph analysis [10], software metrics [11], and formal concept analysis [12], [13]. One drawback however is that, since bottom-up approaches do not start from a model of the required services, the granularity and functionality of the services identified depends on the underlying technology used, and hence may not match the granularity and functionality as required by the architect. In our approach, while we have primarily followed a top down approach, we have used a combination of information retrieval and static analysis to improve the precision.

In the top-down service mining, which is the approach followed in this paper, the focus is on the business domain and on identifying functionality in the monolithic code matching abstract service descriptions in the model created by the business domain experts. This involves matching the natural language descriptions in the model with source code artifacts. This comes under the purview of the areas of concept assignment and feature location, which predominantly use information retrieval (IR) techniques for locating source code matching a given description expressed in domain vocabulary [14], [15], [16], [17], [18], [19], [20], [21]. Of these, the techniques given by Carey and Gannod [15], Zhao et al. [20], and Sindhgatta et

al. [21] are the ones most closely related to our work; therefore we briefly discuss their work here.

Carey and Gannod [15] use metrics along with machine learning methods to create classifiers for differentiating business related service implementations and other services. Zhao et al. [20] argue, like us, that pure IR based technique yield imprecise matching between the concepts and the source code, and propose static code analysis, specifically the use of branch-reserving call graph, to recover relevant and specific code segments for each concept. A similar approach is followed by Sindhgatta et al. [21]. Our contributions are orthogonal to the ones in these papers, in the sense that our primary focus has been a detailed, precise manual case study, and not an evaluation of a primarily automated approach.

Eisenbarth et al. [22] use dynamic analysis along with static analysis for a locating features in source code. The execution trace is obtained by running a set of scenarios invoking the features and analyzed to identify the source code pertaining to the feature. Our experience has been that identifying service implementations ought to be done statically, and that the UI ought to be used for automation. We provide arguments to justify our claim.

Extracting a minimum subset of the program files that not only implement a given domain service but is also executable is useful in wrapping the service or exercising the service functionality independent from the rest of the software system. Herman et al. [14] have proposed unifying slicing with concept assignment to obtain a executable slice of the service implementation. Harman et al. approach of unifying slicing and concept assignment is based on the output of the Hypothesis-Based Concept Assignment (HB-CA) which in turn depends on the *evidence* provided by the domain experts. Our work is complementary, in that the heuristics presented in section V point out potentially useful evidence mechanisms for locating service matches. Also, our goal is to locate only the files that directly implement a collection/service, and not to generate an executable slice.

VII. FUTURE WORK

In the future we would like to undertake more case studies to evaluate the performance of our methodology and heuristics using larger, more complete domain models, as well as larger applications. We will try to reduce subjectivity by having independent experts review the results of our automated heuristics.

Parts of the methodology like seed-finding and expansion need to be better understood, and automated further. We intend to explore program analysis techniques to recover richer features from the code, and to explore information-retrieval techniques to both reduce the burden on humans to provide inputs (e.g., the features for matching, and the various tuning parameters of the algorithm), and to resolve ambiguities in finding the seeds of and fixing the boundaries of collection- and service-implementations. Finally, we would like to study alternative techniques reported in the literature, and to see if we can incorporate those techniques as well as our own in a combined system (e.g., in a probabilistic belief system).

ACKNOWLEDGMENT

We thank Srinivas Padmanabhuni of Infosys Technologies Ltd., India, for valuable suggestions.

REFERENCES

- [1] 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 2008), 12th European Conf. on Softw. Maint. and Reengineering (CSMR 2008), Athens, Greece, 2008*.
- [2] M. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. Kramer, "Service-oriented computing: A research roadmap," *Intl. Journal of Cooperative Inf. Systems*, vol. 17, no. 2, pp. 223–255, 2008.
- [3] A. Arsanjani and A. Allam, "Service-oriented modeling and architecture for realization of an SOA," in *IEEE Intl. Conf. on Services Computing (SCC)*, 2006, p. 521.
- [4] F. Ricca and A. Marchetto, "A 'quick and dirty' meet-in-the-middle approach for migrating to soa," in *IWPSE-Evol '09: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*. New York, NY, USA: ACM, 2009, pp. 73–78.
- [5] H. S. Gupta, D. D'Souza, R. Komondoor, and G. M. Rama, "A case study in matching service descriptions to implementations in an existing system," *CoRR*, vol. abs/1008.2458, August 2010.
- [6] "JAllInOne," <http://jallinone.sourceforge.net>.
- [7] "OpenSwing," <http://oswing.sourceforge.net>.
- [8] S. Rugaber, K. Stirewalt, and L. M. Wills, "The interleaving problem in program understanding," in *Working Conf. in Reverse Engg.*, 1995, pp. 166–175.
- [9] Z. Zhang, R. Liu, and H. Yang, "Service identification and packaging in service oriented reengineering," in *Proc. 17th Intl. Conf. on Softw. Engg. and Knowledge Engg.*, 2005, pp. 620–625.
- [10] S. Li and L. Tahvildari, "A service-oriented componentization framework for java software systems," in *Proc. 13th Working Conf. on Reverse Engg. (WCRE)*, 2006, pp. 115–124.
- [11] G. Caldiera and V. R. Basili, "Identifying and qualifying reusable software components," *IEEE Computer*, vol. 24, no. 2, pp. 61–70, 1991.
- [12] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo, "A method to re-organize legacy systems via concept analysis," in *Proc. 9th Intl. Workshop on Program Comprehension (IWPC)*, 2001, pp. 281–290.
- [13] C. Del Grosso, M. Di Penta, and I. G.-R. de Guzman, "An approach for mining services in database oriented applications," in *Proc. 11th European Conf. on Softw. Maint. and Reengineering*, 2007, pp. 287–296.
- [14] M. Harman, N. Gold, R. M. Hierons, and D. Binkley, "Code Extraction Algorithms which Unify Slicing and Concept Assignment," in *Proc. Working Conf. on Reverse Engg.*, 2002, pp. 11–21.
- [15] M. M. Carey and G. C. Gannod, "Recovering concepts from source code with automated concept identification," in *Proc. 15th IEEE Intl. Conf. on Program Comprehension*, 2007, pp. 27–36.
- [16] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in IR-based concept location," in *2009 IEEE Intl. Conf. on Softw. Maint.*, 2009, pp. 351–360.
- [17] A. Marcus, A. Sergeev, V. Rajlich, and J. I. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," *Working Conf. on Reverse Engg.*, 2004.
- [18] V. Rajlich and N. Wilde, "The Role of Concepts in Program Comprehension," in *Intl. Conf. on Program Comprehension*, 2002, pp. 271–280.
- [19] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *ACM Trans. Softw. Engg. and Methodology (TOSEM)*, vol. 16, no. 1, 2007.
- [20] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "Sniafl: Towards a static noninteractive approach to feature location," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 2, pp. 195–226, 2006.
- [21] R. Sindhgatta and K. Ponnalagu, "Locating Components Realizing Services in Existing Systems," in *IEEE Intl. Conf. on Services Computing (SCC)*, 2008, pp. 127–134.
- [22] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Trans. Softw. Engg.*, vol. 29, no. 3, pp. 210–224, 2003.