Toward Obtaining Event Logs from Legacy Code

Ricardo Pérez-Castillo[1], Barbara Weber[2], Ignacio García-Rodríguez de Guzmán[1]
and Mario Piattini[1]

[1] Alarcos Research Group, University of Castilla-La Mancha
Paseo de la Universidad, 4 13071, Ciudad Real, Spain
{ricardo.pdelcastillo, ignacio.grodriguez, mario.piattini} and@uclm.es
[2] University of Innsbruck
Technikerstraße 21a, 6020 Innsbruck, Austria
barbara.weber@uibk.ac.at

Abstract. Information systems are ageing over time and become legacy information systems which often embed business knowledge that is not present in any other artifact. This embedded knowledge must be preserved to align the modernized versions of the legacy systems with the current business processes of an organization. Process mining is a powerful tool to discover and preserve business knowledge. Most process mining techniques and tools use event logs, registered during execution of process-aware information systems, as the key source of knowledge. Unfortunately, the majority of traditional information systems is not process-aware and does not have any built-in logging mechanisms. Thus, this paper proposes a novel technique to obtain event logs from traditional systems addressing five key challenges. The technique statically analyzes the source code and modifies the source code in a non-invasive manner. The modified source code enables the event registration at runtime based on dynamic source code analysis. The main contribution of this proposal is that the efforts made in the process mining field in terms of tools and mining algorithms can be applied to event logs obtained from traditional information systems.

Keywords. Process Mining, Event Log, Dynamic Analysis, Legacy System

1 Introduction

Business processes have become a key asset in organizations, since processes allow them to know and control their daily performance, and to improve their competitiveness [2]. Thereby, information systems automate most of the business processes of an organization [18]. However, due to uncontrolled maintenance information systems are ageing over time and become legacy systems [15]. They gradually embed meaningful business knowledge that is not present in any other asset of the organization [10]. When maintainability of legacy systems diminishes below acceptable limits, they must be modernized, i.e., the legacy systems are replaced by improved versions [11]. To ensure that the new system is aligned with the organization's business processes, the business knowledge embedded in the information system needs to be preserved [7]. Since the information system is often the only asset of the organization where the business knowledge is present [10], its modernization requires an in-depth understanding of how it currently supports the organization's business processes.

This problem motivates the use of process mining, which became a powerful tool to understand what is really going on in an organization by observing the information systems from three different perspectives [16]: (i) the process perspective focusing on the control flow between business activities; (ii) the organizational perspective describing the organizational structure; and (iii) the case perspective focusing on the characterization of each execution of the process, also known as process instances.

Usually, event logs are obtained from Process-Aware Information Systems (PAIS) [4], i.e., process management systems such as Enterprise Resource Planning (ERP) or Customer Relationship Management (CRM) systems. The nature of these systems (in particular their process-awareness) facilitates the registration of events throughout process execution. Indeed, most process mining techniques and tools are developed for this kind of information systems [2]. In addition to PAIS, there is a vast amount of traditional systems that also support the business processes of an organization, and could thus benefit from process mining. Nevertheless, non process-aware systems imply five key challenges for obtain meaningful event logs: (i) process definitions are implicit described in legacy code and, thus, it is not obvious which events should be recorded in the event log; (ii) the granularity of callable units of an information system and activities of a business process often differs; (iii) legacy code not only contains business activities, but also technical aspects which have to be discarded when mining a business process; (iv) since traditional systems do not explicitly define processes, it has to be established when a process starts and ends; (v) finally, due to the missing process-awareness, it is not obvious how business activities and process instances should be correlated.

This paper proposes a technique for addressing the above mentioned challenges and for obtaining process event logs from traditional (non process-aware) information systems. The technique is based on both static and dynamic analysis of the source code of the systems. Firstly, the static analysis syntactically analyzes the source code and injects pieces of source code in a non-invasive way in specific parts of the system. Secondly, the dynamic analysis of the modified source code makes it possible to write an event log file in MXML format during system execution. The proposed technique is further supported by specific information provided by business experts and system analysts who know the system. The feasibility of our approach is demonstrated with an example based on a simple *Java*-based application for order management. The results obtained from the example show that the proposed technique is able to semi-automatically obtain event logs with a certain quality level.

The remainder of this paper is organized as follows. Section 2 introduces an example to illustrate the challenges as well as the proposed solution. Section 3 introduces the main challenges for obtaining event logs from traditional information systems. Section 4 then presents the proposed technique to tackle these challenges. Section 5 provides an evaluation of the performed example. Section 6 discusses related work and finally, Section 7 provides a conclusion and discusses future work.

2 A Demonstrative Example

This section introduces an example which is used in Section 3 to illustrate all the challenges and demonstrate the feasibility of the proposed technique. The example considers a small business process as the object of study, and a *Java* application implementing the respective process. Fig. 1 shows the source business process which is based on the product order process described by *Weske* [18]. This process allows registered customers to place orders. In parallel, customers receive the products and the invoice to pay the products.

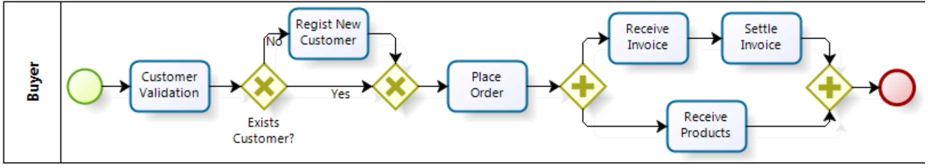
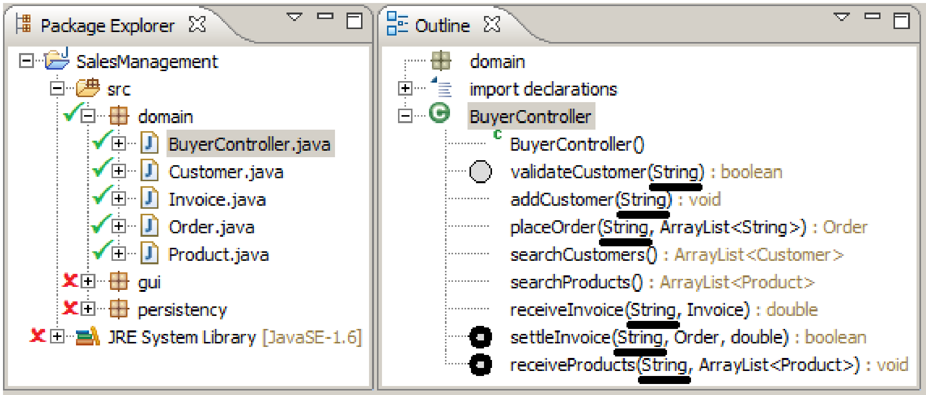


Fig. 1. The source business process for ordering products.



SalesManagement
src
✓ domain
✓ BuyerController.java
✓ Customer.java
✓ Invoice.java
✓ Order.java
✓ Product.java
✗ gui
✗ persistency
✗ JRE System Library [JavaSE-1.6]

domain
import declarations
BuyerController
BuyerController()
○ validateCustomer(String) : boolean
addCustomer(String) : void
placeOrder(String, ArrayList<String>) : Order
searchCustomers() : ArrayList<Customer>
searchProducts() : ArrayList<Product>
receiveInvoice(String, Invoice) : double
● settleInvoice(String, Order, double) : boolean
● receiveProducts(String, ArrayList<Product>) : void

Fig. 2. Structure of the source *Java* application.

Fig. 2 shows the structure of the small application developed to support the source business process. The application follows the traditional decomposition into three layers [5]: (i) the *domain* layer supports all the business entities and controllers; (ii) the *presentation* layer deals with the user interfaces; and (iii) the *persistency* layer handles the data access (see Fig. 2 left). The *BuyerController* class contains most of the logic of the application (see Fig. 2 right), i.e., it provides the methods that support the activities of the source business process.

3 Process-Awareness Challenges

This section shows the main challenges for obtaining event logs from traditional information systems: missing process-awareness, granularity, discarding technical code, process scope, and process instance scope.

3.1 Challenge 1 - Missing Process-Awareness

Knowing what activities are executed is the first important challenge for registering the events of a traditional (non process-aware) information system. This problem is caused by the different nature of traditional information systems and PAIS. While

PAISs manage processes that consist of a sequence of activities or tasks with a common business goal using explicit process descriptions [18] (see Fig. 3A), traditional systems are a set of methods, functions or procedures (*callable units* in general) where processes are only implicitly described and thus blurred. Traditional systems can be seen as a graph where the nodes are the different callable units, and the arcs are the calls between callable units (see Fig. 3B). Thereby, the call graph represents the control flow of a traditional system according to the domain logic implemented.

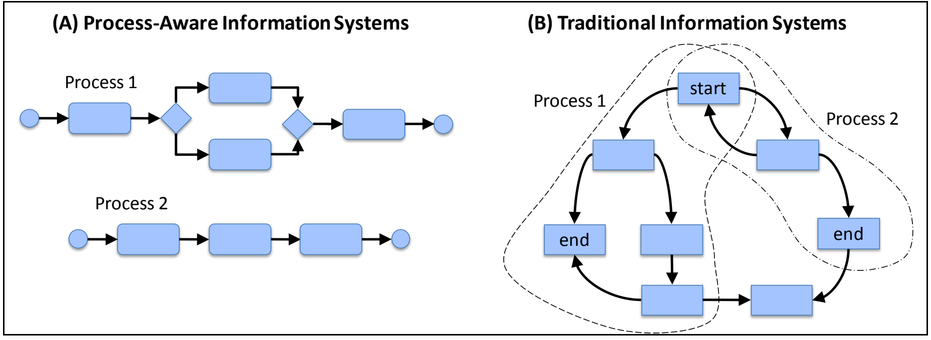

(A) Process-Aware Information Systems
Process 1
Process 2
(B) Traditional Information Systems
Process 1
start
Process 2
end
end

Fig. 3. Comparison between PAIS and traditional information systems

To address this challenge *Zou et al.* [19] proposed the *"a callable unit / a business activity"* approach which considers each callable unit of a traditional system as a candidate business activity in a process mining context. Although this approach provides a good starting point, it ignores several important challenges related to the inherent nature of source code such as, for example, the different granularity of callable units and business activities (cf. Section 3.2) and the mixture of business related callable units and technical callable units which are typical for legacy information systems (cf. Section 3.3).

3.2 Challenge 2 - Granularity

The different granularity of business activities and callable units in legacy systems constitutes another important challenge. In [12], each callable unit in a traditional legacy system is considered as an activity to be registered in an event log. However, traditional systems typically contain thousands of callable units. While some of them are large callable units supporting the main business functionalities of the system, many callable units are very small and do not directly support any business activity. In the example, all *setter* and *getter* methods of the classes representing business entities like *Customer* or *Product* (see Fig. 2) only read or write object fields and thus can be considered as fine-grained units. To avoid that the mined business processes get bloated with unnecessary details, too fine-grained callable units should not be considered as activities in the event log, but be discarded. Unfortunately, the set of callable units cannot easily be divided into coarse-and fine-grained callable units, since the threshold between these subsets is unknown.

In this sense, different solutions can be implemented to discard fine-grained callable units. On the one hand, source code metrics (such as the lines of source code metric or the cyclomatic complexity metric) could be used to determine if a callable unit is a coarse- or fine-grained unit [14]. This solution is easy to implement, but has the disadvantage of high computational costs when the event log file is written during run time. On the other hand, heuristics (like discarding getter and setter methods, or discarding units when call hierarchies reach a specific depth) could offer a good alternative with minimal computational costs.

3.3 Challenge 3 - Discarding Technical Code

Another important challenge is caused by the fact that legacy information systems typically contain several callable units, which cannot be considered as business activities. Callable units can be grouped into two groups: (i) the *problem domain* group contains the callable units related to the business entities and functionalities of the system to solve the specific problem (i.e., these units implement the business processes of the organization) and (ii) the *solution domain* group contains the callable units related to the technical nature of the used platform or programming language and aids the callable units of the previous group. Since callable units belonging to the solution domain do not constitute business activities, they should not be considered in the event log.

However, *how can we know whether or not a callable unit belongs to the solution domain?* As a first approximation callable units in charge of auxiliary or technical functions that are not related to any use case of the system (e.g., callable units belonging to the *presentation* or *persistency* layer in the example) can be discarded. However, due to the delocalization and interleaving problems [13] the problem and solution domain groups are not always disjoint sets (i.e., a domain package can contain some technical units or a technical package can contain some domain code). In the example, the methods *searchCustomers* and *searchProducts* in the class *BuyerController* (see Fig. 2) mix problem and solution code, since these methods also contain code related to database access. As a consequence, in many cases the only possible solution is that system analysts provide the information about whether a callable unit belongs to the problem or solution domain.

3.4 Challenge 4 - Process Scope

Another important challenge is to establish the scope of a business process (i.e., to identify where a process instance starts and ends). While the start and end points of a business process are explicitly defined in PAISs (see Fig. 3A), traditional information systems lack any explicit information about the supported processes (see Fig. 3B).

Unfortunately, the information where a process starts and ends cannot be automatically derived from the source code. In the example, there is not enough information to derive what methods support the start and end points of the source business process. Therefore, this information must be provided by business experts and system analysts. On the one hand, business experts know the business processes of the organization as well as their start and end activities. On the other hand, system analysts know what callable units in the source code support the start and end activities.

3.5 Challenge 5 - Process Instance Scope

The lack of process-awareness in traditional information systems causes another fundamental challenge which is due to the fact that a business process is typically not only executed once, but multiple instances are executed concurrently. If a particular business activity is executed (i.e., callable unit is invoked), this particular event has to be correctly linked to one of the running process instances. For example, imagine the source business process of the example (see Fig. 1). The Java application supporting that business process could execute the sequence *'Customer Validation'* (Customer 1), *'Customer Validation'* (Customer 2), *'Place Order'* (Customer 1) and *'Place Order'* (Customer 2) for two different customers. To obtain meaningful event logs, the activities which are executed by the information system need to be correctly linked to either Customer 1 or Customer 2 (i.e., the customer information in this example uniquely identifies a process instance).

Correlating an activity with a data set, which uniquely identifies the process instance it belongs to (e.g., the customer name), poses significant challenges. In particular, it has to be established which objects can be used for uniquely identifying a process instance (i.e., what the correlation data is). If correlation objects have been identified, the location of these objects in each callable unit has to be determined (i.e., the argument or variable in each callable unit that contains the correlation data). This requires the input of business experts and systems analysts who know the information system and the process it supports. Unfortunately, however, there are some methods (e.g. *searchCustomers* or *searchProducts* in the example) where the selected correlation data does not exist. For this reason, traceability mechanisms throughout callable units need to be implemented to have the correlation data available at any time.

4 The Proposed Solution

This paper proposes a technique to obtain event logs from non process-aware systems addressing the previously discussed challenges. Our proposal presents the guidelines of a generic technique, although it is specially designed for object-oriented systems.

The technique is based on dynamic analysis of source code combined with a static analysis. The *static analysis* examines the source code in a static way, and modifies the source code by injecting code for writing specific events during its execution (cf. Section 4.1). After static analysis, the source code is *dynamically analyzed* at runtime by means of the injected sentences (cf. Section 4.2). Fig. 4 gives an overview of the technique, the tasks carried out and the artifacts obtained (gray color).

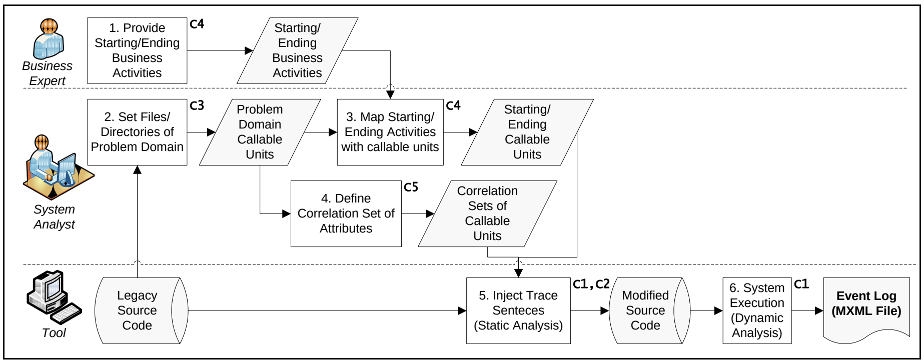

Business Expert
1. Provide Starting/Ending Business Activities
C4
Starting/ Ending Business Activities
System Analyst
2. Set Files/ Directories of Problem Domain
C3
Problem Domain Callable Units
3. Map Starting/ Ending Activities with callable units
C4
Starting/ Ending Callable Units
4. Define Correlation Set of Attributes
C5
Correlation Sets of Callable Units
Tool
Legacy Source Code
5. Inject Trace Senteces (Static Analysis)
C1,C2
Modified Source Code
6. System Execution (Dynamic Analysis)
C1
Event Log (MXML File)

Fig. 4. The overall process carried out by means of the proposed technique

4.1 Static Analysis for Injecting Source Code

The static analysis is the key stage of the proposed technique, where special sentences for writing events during system execution are injected in the source code. Due to the missing process-awareness of traditional information systems this task poses several challenges (as introduced in Section 3). While challenges C1 and C2 can be addressed in a fully automated manner (Task 5 and 6 in Fig. 4), challenges C3, C4 and C5 require input from the business expert and /or the system analyst (Task 1 - 4 in Fig. 4).

In the first task, business experts establish the start and end business activities of the business processes to be discovered. This information is necessary to deal with the process scope challenge (Challenge C4). In parallel, system analysts examine in the second task the legacy source code and filter the directories, files or set of callable units that support business activities, (i.e., they select the callable units belonging to the problem domain). This information is necessary to reduce potential noise in the event log due to technical source code (Challenge C3). The third task is the mapping between start/end business activities and the callable units supporting them, which is again supported by the system analysts (Challenge C4).

In the fourth task system analysts establish the correlation data set for each callable unit which is uniquely identifying a process instance (Challenge C5). For this, the correlation data is mapped to one or more parameters of each callable unit. This information is then used during run-time when the dynamic analysis writes the event log to correlate the executed activities with the proper process instance. Unfortunately, the mapping of correlation data and parameters of callable units is not always feasible, since the correlation data is not available in all intermediate or auxiliary callable units. In order to solve this problem, the technique chooses a heuristic solution that includes, whenever the correlation data is empty, callable units without correlation data in the same process instance than the last executed callable unit. This solution is implemented during the final dynamic analysis at run time (cf. Section 4.2).

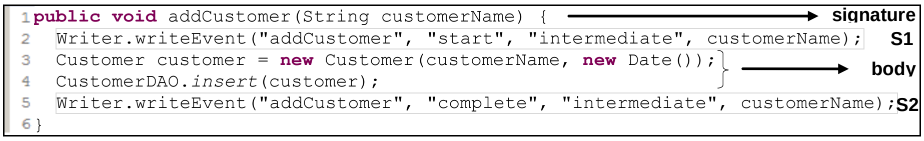
Fig. 2 shows the information provided by the system analysts for mining the order product process from the example. The files or directories that do not contain technical source code, and therefore belong to the problem domain, are marked with a tick (Task 2). The methods that support start or end activities are marked with circles, whereby thin lines are used to represent start activities and thick lines for end activities (Task 1, 3). In this example, the customer name is used as correlation data, because it allows for uniquely identifying process instances for that particular application (Task 4). For this, the system analyst defines the correlation data by selecting the respective parameters of the callable units (in Fig. 2, underlined *string* parameters that contain the customer name information).

After that, the fifth task consists of the syntactic analysis of the source code. The parser analyzes and injects on the fly the special sentences writing the event long during system execution. This analysis can be automated following the algorithm presented in Fig. 5. During the static analysis, the source code is broken down into callable units (Challenge 1), and then, the algorithm only modifies the units that belong to the problem domain subgroup selected by the system analyst in Task 3 (Challenge 3). In addition, fine-grained callable units (e.g., *setter*, *getter*, *constructor*, *toString* and *equals* callable units) are automatically discarded (Challenge C2). Finally, in each of the filtered callable units, two sentences are injected at the beginning and the end of each respective unit. The first sentence represents an event with a *start* event type, and the second one represents the *complete* event for the same business activity. Moreover, the correlation data defined for the unit as well as information whether or not the unit represents a start or end activity are included in the sentences. When the modified code is executed, the injected sentences invoke the *WriteEvent* function, which writes the respective event in the event log (for details see Section 4.2).

```
InjectTraces (CallableUnits, ProblemDomainCallableUnits, StartingCallableUnits, EndingCallableUnits)
    ModifiedCallableUnits ← ϕ
    c' ← null
    For (c ∈ CallableUnits)
        If (c ∈ ProblemDomainCallableUnits)
            If (c ∈ StartingCallableUnits)
                position ← "first"
            Else If (c ∈ EndingCallableUnits)
                position ← "last"
            Else
                position ← "intermediate"
            sentence$_1$ ← WriteEvent (c.name, "start", position, c.correlationSet)
            sentence$_2$ ← WriteEvent (c.name, "complete", position, c.correlationSet)
            c'.signature ← c.signature
            c'.body ← sentence$_1$ + c.body + sentence$_2$
            ModifiedCallableUnits ← ModifiedCallableUnits ∪ {c'}
        Else
            ModifiedCallableUnits ← ModifiedCallableUnits ∪ {c}
    Return ModifiedCallableUnits
```

Fig. 5. Algorithm to inject traces by means of static analysis.

Continuing the example, Fig. 6 shows the method *addCustomer* after the injection of the special sentences. According to the algorithm (see Fig. 5) the sentence S1 is added directly after the method signature. The body of the source method is then added without any changes and finally sentence S2 is added after the body to the method.



```
public void addCustomer(String customerName) {
  Writer.writeEvent("addCustomer", "start", "intermediate", customerName);
  Customer customer = new Customer(customerName, new Date());
  CustomerDAO.insert(customer);
  Writer.writeEvent("addCustomer", "complete", "intermediate", customerName);
}
```

Fig. 6. The *Java* method *'addCustomer'* modified with the injected sentences.

4.2 Dynamic Analysis for Obtaining Event Logs

After static analysis the modified source code can be released to production again. The new code makes it possible to write event log files according to the MXML (Mining XML) format [6], which is used by the process mining tool *ProM* [17]. When the control flow of the information system reaches an injected sentence, a new event is added to the event log. The events are written by means of the *WriteEvent* function. The parameters of the function are: (i) the name of the executed callable unit; (ii) the event type (start or complete); (iii) the position of the activity that represents the executed unit (first, intermediate or last); and (iv) the correlation data to uniquely identify each process instance. These parameters are established during static analysis, although the correlation data is only known at runtime.

To add a new entry to the log file the function starts searching the adequate process of the event log where the event must be written by means of an *Xpath* expression [3]. If the process is null, then a new process is created. After that, the function examines the correlation data to determine to which process instance the event has to be added. If the correlation data is empty, then the algorithm takes the correlation data of the previously executed callable unit to add the event to the correct process instance. This solution is based on simple heuristics and allows correlating events and process instances when no correlation data is available for the respective event. Moreover, in order to add the event to the correct process instance, the *WriteEvent* function again uses an *Xpath* expression taking the correlation data into account. If the expression does not find a process instance for the correlation data (i.e., because the event belongs to a start activity), the function creates a new process instance for the correlation data.

Finally, when the function has determined the correct process instance, it adds the event to that particular instance. The event, represented as an *AuditTrailEntry* element in an MXML file [6], is created with (i) the name of the executed callable unit that represent the *WorkflowModelElement*; (ii) the event type that is also a parameter of the algorithm; (iii) the user of the system that executed the callable unit (or the user of the session if the system is a web application), which represents the *originator* element; and finally (iv) the system date and time when the callable unit was executed to represent the *timestamp* element.

5 Evaluation

According to the example, after source code modification, the modified application is released to production. In order to evaluate the obtained event log in a controlled way, a finite set of transactions is executed with the modified application. The customers involved in the transactions are *John Doe* (a registered customer) as well as *Jane Doe* and *Foo* (two unregistered users). Fig. 7 shows the transactions which were carried out.

```
(1) Jane Doe buys pdt1 and pdt2      (2) John Doe buys pdt1
(3) Jane Doe buys pdt3 and pdt4      (4) Foo buys pdt2, pdt3 and pdt4
(5) John Doe buys pdt4               (6) Foo buys pdt1
```

Fig. 7. The set of transactions executed in the example.

After the application of the proposed technique, the event log is analyzed by means of *ProM* [17], to check whether the obtained result is aligned with the source business process. The obtained log contains 1 process with 6 process instances. The log has in total 78 events and between 12 and 16 of events per process instance. The examination of the event log reveals that each process instance is related to a specific transaction executed in the application. In a next step, the genetic mining plugin of *ProM* [9] is used for process discovery. Fig. 8 shows the discovered business process considering a population size of 100 and using 1000 generations in the genetic algorithm. The discovered business process has a fitness value of 0.94.

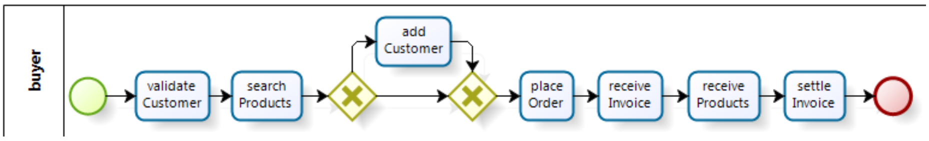


Fig. 8. The business process discovered by means of the genetic algorithm of *ProM*.

The comparison between the source and discovered business process shows some deviations. The first difference is related to the activity names, i.e., the names in the discovered process are inherited from the source code and therefore slightly differ from the labels used in the source business process. Another important difference is that in the discovered business process activity *'searchProducts'* (which is not present in the source process) occurs after activity *'validateCustomer'*. This deviation results from a technical method that was not filtered out by the system analyst during static analysis. Finally, the parallel branches at the end of the process are not mined correctly (i.e., activities *'receiveInvoice'*, *'receiveProducts'* and *'settleInvoice'* are carried out sequentially, instead of concurrently). This deviation is due to the fact that the operations are always executed in the same order through the application that supports the source business process. Despite these deviations, the obtained process gives a good starting point to understand the source business process. In addition, the technique can be applied iteratively, i.e., business experts and system analysts can refine the provided information in order to obtain event logs representing the business process more accurately.

6 Related Work

Related to our approach is existing work on the recovery of business processes from non process-aware information systems. *Zou et al* [19] developed a framework to recover workflows from legacy information systems. This framework statically analyzes the source code and applies a set of heuristic rules to discover business knowledge from source code. *Pérez-Castillo et al* [12] make another proposal based on static analysis that uses a set of business patterns to discover business processes from source code. Both approaches solely rely on static analysis, which has the disadvantage that activities cannot be linked correctly to process instances, since the required correlation data is only known at runtime. Thus, other solutions based on dynamic analysis have been suggested. *Cai et al.* [1] propose an approach that combines requirement reacquisition with dynamic analysis. Firstly, a set of use cases is recovered by means of interviewing the system's users. Secondly, the system is dynamically traced based on these use cases to recover business processes. In all these works, the technique for recovering event logs is restricted to a specific mining algorithm. In contrast, our solution proposes a technique based on dynamic analysis (combined with static analysis) to obtain MXML event logs from traditional information systems that is not restricted to a specific process mining algorithm. Similar to our approach the work of *Ingvaldsen et al.* [8] aims at obtaining logs in MXML format from ERP systems. Thereby, they consider the SAP transaction data to obtain event logs. In contrast, our approach aims at traditional information systems without any built-in logging features. In addition, *Günther et al.* [6] provide a generic import framework for obtaining MXML event logs from different PAISs.

7 Conclusions and Future Work

This paper presents a novel technique based on static and dynamic analysis of source code to obtain event logs from non process-aware systems. Thereby, the obtained event log can be used to discover business processes in the same way than an event log obtained from any PAIS. Thus, all the research and development efforts carried out in the process mining field may be exploited for traditional information systems. Achieving this goal is very ambitious since at least five key challenges must be addressed: (i) missing process-awareness, (ii) granularity, (iii) discarding technical code, (iv) process scope and (v) process instance scope.

In a first step, the proposed technique applies static analysis for injecting special sentences in the source code. In a second step, the modified source code is executed, and an event log is written during system execution. A demonstrative example illustrates the feasibility of the proposal.

In principle, the static analysis of the system has to be performed only once, and then the modified source code can be dynamically analyzed several times to obtain different event logs. However, the feedback obtained by business experts and systems analysts, after the first static and dynamic analysis, can be used to incrementally refine the next static analysis for improving the results obtained during dynamic analysis.

Our future work will focus on the improvement of the proposed technique. A traceability mechanism will be implemented taking the call hierarchies into account to deal with lost and scattered correlation data. In addition, in order to accurately detect the strengths and weakness, the proposal will be validated by means of a case study involving a real-life information system.

Acknowledgement

This work was supported by the FPU Spanish Program; by the R+D projects funded by JCCM: ALTAMIRA (PII2I09-0106-2463), INGENIO (PAC08-0154-9262) and PRALIN (PAC08-0121-1374); and the PEGASO/MAGO project (TIN2009-13718-C02-01) funded by MICINN and FEDER. In addition, this work was supported by the Quality Engineering group at the University of Innsbruck.

References

[1] Cai, Z., X. Yang, and W. Wang, Business Process Recovery for System Maintenance - An Empirical Approach, in 25 th International Conference on Software Maintenance (ICSM'09). 2009, IEEE CS: Edmonton, Canada. p. 399-402.

[2] Castellanos, M., K.A.d. Medeiros, J. Mendling, B. Weber, and A.J.M.M. Weitjers, Business Process Intelligence, in Handbook of Research on Business Process Modeling, J. J. Cardoso and W.M.P. van der Aalst, Editors. 2009, Idea Group Inc. p. 456-480.

[3] Clark, J. and S. DeRose, XML Path Language (XPath). 1999, World Wide Web Consortium (W3C).

[4] Dumas, M., W. van der Aalst, and A. Ter Hofstede, Process-aware information systems: bridging people and software through process technology. 2005: John Wiley & Sons, Inc.

[5] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Longman Publishing Co. ed. 1995, Inc. Boston, MA, USA: Addison Wesley.

[6] Günther, C.W. and W.M.P. van der Aalst, A Generic Import Framework for Process Event Logs. Business Process Intelligence Workshop (BPI'06), 2007. LNCS 4103: p. 81-92.

[7] Heuvel, W.-J.v.d., Aligning Modern Business Processes and Legacy Systems: A Component-Based Perspective (Cooperative Information Systems). 2006: The MIT Press.

[8] Ingvaldsen, J.E. and J.A. Gulla, Preprocessing Support for Large Scale Process Mining of SAP Transactions. Business Process Intelligence Workshop (BPI'07) 2008. LNCS 4928: p. 30-41.

[9] Medeiros, A.K., A.J. Weijters, and W.M. Aalst, Genetic process mining: an experimental evaluation. Data Min. Knowl. Discov., 2007. 14(2): p. 245-304.

[10] Mens, T., Introduction and Roadmap: History and Challenges of Software Evolution Software Evolution (Springer Berlin Heidelberg), 2008. 1: p. 1-11.

[11] Newcomb, P., Architecture-Driven Modernization (ADM), in Proceedings of the 12th Working Conference on Reverse Engineering. 2005, IEEE Computer Society.

[12] Pérez-Castillo, R., I. García-Rodríguez de Guzmán, O. Ávila-García, and M. Piattini, MARBLE: A Modernization Approach for Recovering Business Processes from Legacy Systems, in International Workshop on Reverse Engineering Models from Software Artifacts (REM'09). 2009, Simula Research Laboratory Reports: Lille, France. p. 17-20.

[13] Ratiu, D., Reverse Engineering Domain Models from Source Code, in International Workshop on Reverse Engineering Models from Software Artifacts (REM'09). 2009, Simula Research Laboratory: Lille, France. p. 13-16.

[14] Rozman, I., J. Györkös, and T. Dogsa, Relation Between Source Code Metrics and Structure Analysis Metrics, in Proceedings of the 3rd European Software Engineering Conference. 1991, Springer-Verlag. p. 332-342.

[15] Ulrich, W.M., Legacy Systems: Transformation Strategies. 2002: Prentice Hall. 448.

[16] van der Aalst, W. and A.J.M.M. Weijters, Process Mining, in Process-aware information systems: bridging people and software through process technology, M. Dumas, W. van der Aalst, and A. Ter Hofstede, Editors. 2005, John Wiley & Sons, Inc. p. 235-255.

[17] van der Aalst, W.M.P., B.F. van Dongenm, C. Günther, A. Rozinat, H.M.W. Verbeek, and A.J.M.M. Weijters, ProM : the process mining toolkit, in 7th International Conference on Business Process Management (BPM'09) - Demonstration Track. 2009: Ulm, Germany. p. 1-4.

[18] Weske, M., Business Process Management: Concepts, Languages, Architectures. 2007, Leipzig, Alemania: Springer-Verlag Berlin Heidelberg. 368.

[19] Zou, Y. and M. Hung, An Approach for Extracting Workflows from E-Commerce Applications, in Proceedings of the Fourteenth International Conference on Program Comprehension. 2006, IEEE Computer Society. p. 127-136.