Deriving Event Logs from Legacy Software Systems

Marius Breitmayer[1][0000−0003−1572−4573], Lisa Arnold[1][0000−0002−2358−2571], Stephan La Rocca[2], and Manfred Reichert[1][0000−0003−2536−4153]

1 Institute of Databases and Information Systems, Ulm University, Germany
{marius.breitmayer,lisa.arnold,manfred.reichert}@uni-ulm.de
2 PITSS GmbH Stuttgart, Germany {slarocca}@pitss.com

Abstract. The modernization of legacy software systems is one of the key challenges in software industry, which requires comprehensive system analysis. In this context, process mining has proven to be useful for understanding the (business) processes implemented by the legacy software system. However, process mining algorithms are highly dependent on both the quality and existence of suitable event logs. In many scenarios, existing software systems (e.g., legacy applications) do not leverage process engines capable of producing such high-quality event logs, which hampers the application of process mining algorithms. Deriving suitable event log data from legacy software systems, therefore, constitutes a relevant task that fosters data-driven analysis approaches, including process mining, data-based process documentation, and process-centric software migration. This paper presents an approach for deriving event logs from legacy software systems by combining knowledge from source code and corresponding database operations. The goal is to identify relevant business objects as well as to document user and software interactions with them in an event log suitable for process mining.

Keywords: Event Log Generation · Legacy Software System · Software Modernization · Process Mining

1 Introduction

Economically, one of the most important sectors in software industry concerns the modernization of legacy software systems. These systems need to be replaced by modern software systems showing better usability, higher performance, and improved code quality. A successful modernization of a legacy software system requires the analysis of the (business) processes implemented by the legacy software, the interactions users have with the system, and the access points to system information (e.g., source code or databases).

Process mining offers a plethora of analysis approaches to gain a broad understanding of the processes implemented in software systems. Process discovery, for example, enables the derivation of process models from event logs [1]. In turn, conformance checking correlates modeled and recorded behavior of a business process, enabling the analysis of the observed process behavior in relation to a given process model [2]. Finally, process enhancement allows improving business processes based on the information recorded in event logs. In summary, most process mining approaches highly depend on the existence of process event logs as well as the quality of these logs.

In software modernization projects, legacy software systems need to be analyzed. In this context, the use of process mining approaches is very promising for analyzing the processes implemented in these systems. However, most existing legacy software systems neither have been designed based on pre-specified executable process models nor do they provide extensive process logging capabilities. As a consequence, the application of process mining to legacy software systems is hampered and alternatives for obtaining models of the implemented processes and, thus, for supporting the migration of the legacy software system to modern technology are needed. Alternatives include, for example, extensive interviews with key system users and process owners [3]. Both alternatives, however, are time-consuming and prone to incompleteness.

This paper presents an approach to generate event logs from running legacy software systems by combining knowledge from source code analysis, including database statements, to discover the relevant business objects of a process as well as to document user and software interactions in an event log suitable for process mining. We consider the following research questions:

RQ1: How can we generate event logs from running legacy software systems?
RQ2: How can we ensure that the performance of legacy software systems is not affected during event log generation?

The remainder of this paper is structured as follows: Section 2 introduces the concepts necessary for understanding this work. Section 3 discusses the requirements for generating event logs from running legacy software systems. Section 4 presents the legacy software system analysis required for generating event logs. Section 5 describes our approach and shows how one can extend a legacy software system to generate event logs. Section 6 evaluates our work using a requirements evaluation, a performance comparison, and a user survey. Section 7 discusses related work. Section 8 provides a short summary as well as an outlook.

2 Fundamentals

2.1 Legacy Software Systems

Legacy software systems are widespread in enterprises, but very costly to maintain due to bad documentation, outdated operating or development environments, or high complexity of the historically grown system code basis [4]. As a result, the replacement of such legacy software systems is often significantly delayed beyond the initial system lifespan. Legacy software systems consist of a plethora of artifacts and resources such as servers, (non-normalized) databases, source code, or user forms, which all may be used during legacy software system analysis. Fig. 1 depicts a screenshot of an Oracle legacy software system implemented in the 1990s. We will refer to this example in the following.

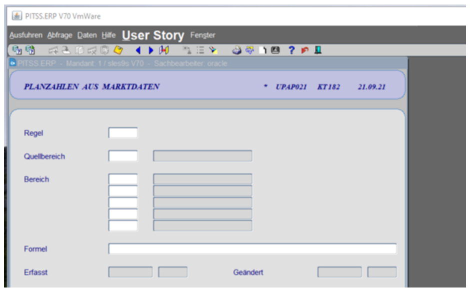

PITSS.ERP V70 VmWare
Ausfuhren Abfrage Daten Hilfe User Story Fenster
PITSS.ERP - Mandant: 1 / sles9s V70 - Sachbearbeiter: oracle
PLANZAHLEN AUS MARKTDATEN
* UPAP021 KT182 21.09.21
Regel
Quellbereich
Bereich
Formel
Erfasst
Geändert

Fig. 1: Screenshot of a Legacy Software System

2.2 Event Logs

Event logs build the foundation for process mining algorithms and capture information on cases, events, and corresponding activities [5]. In general, event logs record events related to the execution of process instances. Mandatory attributes of a log entry include the case identifier, the timestamp, as well as the executed activity [5].

In the context of legacy software systems, which may support multiple process types (e.g., order-to-cash, purchase-to-pay, or checking an invoice), it might be unclear to which process type an activity belongs. Therefore, an additional attribute indicating the process type is required when deriving event logs from legacy software systems.

3 Requirements

In most cases, there exist no suitable event logs for process mining in legacy software systems. This section elicits fundamental requirements to be met when generating event logs from user and software interactions with legacy software systems. On one hand, we gathered the requirements from literature [5]. On the other, we conducted interviews with domain experts (e.g., software engineers, and process owners) to complement these requirements. Amongst others, we identified the following requirements:

Requirement 1: (*Relevance*) The event log should only contain process-relevant data that refers to those interactions with the legacy software system that correspond to a process (e.g., filling or completing a form). If an interaction triggers an automated procedure (e.g., invocation of an operation in the legacy software system), the resulting changes (e.g., to the database) should be recorded in the event log as well.

Requirement 2: (*Scope*) Legacy software systems often use a plethora of database tables and source code fragments that contain business-relevant data. Identifying and scoping process-relevant database tables and code fragments usually requires extensive domain knowledge that might not be available. An approach for generating event logs from legacy software systems should therefore minimize the domain knowledge required.

Requirement 3: (*Consistency*) To facilitate the preprocessing of the event log data, the event log should be consistent with respect to timestamps, data types, and additional resources, even if different software components of the legacy software system (e.g., database and user forms) are involved.

Requirement 4: (*Performance*) The event log generation from a running legacy system should not influence its performance, i.e., the user and software interactions should not be influenced (e.g., due to increased loading times).

4 Legacy Software System Analysis

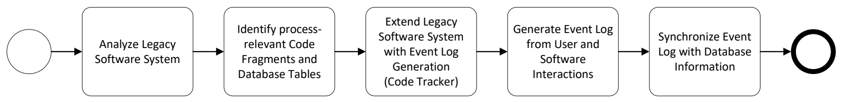

Analyze Legacy Software System
Identify process-relevant Code Fragments and Database Tables
Extend Legacy Software System with Event Log Generation (Code Tracker)
Generate Event Log from User and Software Interactions
Synchronize Event Log with Database Information

Fig. 2: Preparation Steps of our Approach

We derived the approach for generating event logs from running legacy software systems (cf. Fig. 2) by applying design science research [6].

In the first step, we analyze the legacy software system, including source code, database tables, and additional resources (e.g., configuration files, user forms displayed by the running legacy software system).

In the second step, we transform the source code of the legacy application to an abstract syntax tree in order to identify those code elements that trigger database operations (e.g., the selection, insertion, deletion or update of tuples in database tables). Using the database tables in combination with the information provided from the source code (e.g., the exact SQL statement), we address an important problem of legacy software systems, i.e., we are able to identify relations between tables that have not been explicitly specified using foreign-key constraints. In other words, we identify additional relations between database tables specified in the legacy application source code.

We can further build clusters of database tables that most likely belong to the same process based on these identified relations. In Fig. 3, for example, tables belonging to the cluster marked in green correspond to orders, whereas tables of the purple cluster correspond to articles. We identify the center of a cluster using a page rank algorithm [7]. Note that checking the identified clusters with a domain expert (if possible) might further improve the event log generation (cf. Requirements 1 and 2 in Section 3). After having identified the clusters in the database tables, we can determine which source code fragments are relevant for the generation of the event log, i.e., which code fragments affect process-relevant database tables. This information can then be used to configure and install the code tracker into the legacy software system.

The code tracker is able to automatically inject code fragments into the source code, which, in turn, are then executed together with the legacy software system code enabling the generation of event logs at runtime. To ensure that the performance of the legacy software system is not negatively affected, the necessary data is passed using common log mechanisms (e.g., *java.util.logging* or

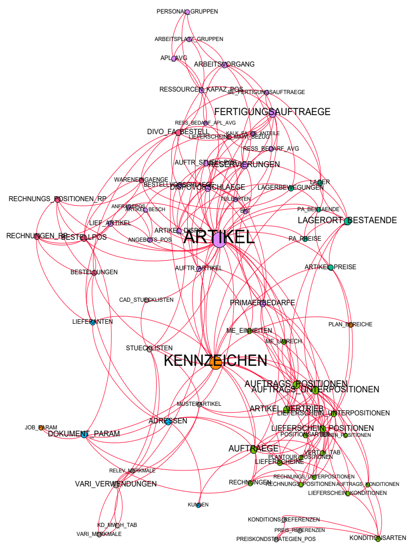


Fig. 3: Clusters derived from Database and Source Code

Oracle message-builtIns) already available in the legacy software system. This yields the advantage that the existing infrastructure, in which the legacy software system operates, takes care of managing files, rotating data and, thus, providing methods for writing data to an event log in a performant manner. Consequently, the transfer of event log data becomes possible with minimal footprint. In a last step, we synchronize the event log with the information from the database (e.g., redo logs) enabling the generation of high-quality event logs.

5 Event Log Generation

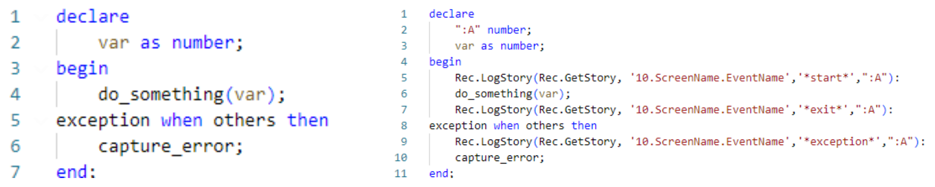
In the context of a legacy software system, a business process can be derived from the sequences of interactions the users have with the legacy application. Each interaction of such a sequence is then subject-bound (i.e., the interactions of a sequence belong to the same transaction). In a legacy software system, such processes may be initiated and terminated using pre-defined actions, for example, menu items or key combinations. The addition of corresponding actions to an event log, together with the associated application object (e.g., a product identified by a unique product number, or an order identified by its order number) constitutes the basis for generating an event log. Subsequently, this event log may then serve as input for process mining algorithms.

5.1 Legacy Software System Extension

After showing how process-relevant source code fragments can be identified in the legacy software system (cf. Section 4), we discuss how to augment the legacy software system with event log generation capabilities by installing the *code* *tracker*. This installation utilizes our ability to parse the relevant source code fragments and to map them as an abstract syntax tree [8].

Leveraging this source code information, we can add the code tracker nodes at the relevant positions of the software code, i.e., “start”, “end”, “return”, “exit”, and “exception”, surrounding a create-, read-, update- or delete-statement (CRUD-statement). Each code tracker statement then captures the context (i.e., the position of the relevant source code in the entire legacy software system), the timestamp, the identifier of the corresponding user session, and, optionally, additional parameters of the identified source code fragments.

Adding the code tracker to the legacy software system is implemented as a pre-deployment task. Thus, no developer interaction becomes necessary. In a deployment chain, relevant code is checked out, parsed, added to the tracker, saved, compiled, and then deployed to the running legacy software system. This integration ensures that any kind of source code change or release of new software versions can be captured, hence, preventing mismatches between the running code and the information captured in the generated event log. As an example, consider the code fragment depicted in Fig. 4a, which is responsible for handling a user interaction event. When applying the code tracking pre-deployment task to this code fragment, we obtain the code fragment depicted in Fig. 4b. In the latter, the event log generation is added to lines 2, 5, 7, and 9. Note that *ScreenName* and *EventName* constitute placeholders that are replaced by the actual values at runtime. An example of such actual values could be ORDERS.MAIN_CANVAS.BUTTON_SAVE.WHEN_BUTTON_PRESSED.



```
declare
    var as number;
begin
    do_something(var);
exception when others then
    capture_error;
end:
```

(a) Source Code

```
1   declare
2       ":A" number;
3       var as number;
4   begin
5       Rec.LogStory(Rec.GetStory, '10.ScreenName.EventName','*start*',":A"):
6       do_something(var);
7       Rec.LogStory(Rec.GetStory, '10.ScreenName.EventName','*exit*',":A"):
8   exception when others then
9       Rec.LogStory(Rec.GetStory, '10.ScreenName.EventName','*exception*',":A"):
10      capture_error;
11  end;
```

(b) Extended Source Code

Fig. 4: Example Source Code Fragments

During event log analysis, such values provide important contextual information and enable a failure-free identification of documented user and software interactions. There exists a plethora of user interactions, e.g., pressing a button, entering a value into a form field, clicking on a check box, or navigating between elements. As long as the legacy software system implements these events as process-relevant in the source code, the code tracker is added.

Merging user interactions with database events In addition to the user interaction events gathered by the code tracker, we analyze all database updates (i.e., insert, update and delete) expressed in terms of Data Manipulation Language (DML) statements. For this analysis, we utilize the redo log capabilities provided by the legacy software system database. Redo logs are created by transactional databases, to enable recovery in case of failures (e.g., after crashes). The information contained in a redo log consists, for each recorded operation, of the name of the database table, the performed operation (i.e., insert, update, or delete), the timestamp, the session-id, and the original DML statement applied to the database [9].

From the source code extension (cf. Fig. 4b), for each event, we can also extract the timestamp, session-id, and the affected database table. Combining these three attributes enables the allocation between user or software interactions and the corresponding changes to the persistence layer of the legacy software system. Leveraging the information from redo logs, again ensures that no performance penalties emerge due to the event log generation.

Using the code tracker functions, the information captured in the event log is significantly increased compared to an event log solely generated from the database schema [10], as we can unambiguously link processes with both program code and related data. Therefore, time-consuming reverse engineering and root cause analysis are not needed as the connection between source code, data, and processes already exists.

Finally, one valuable effect for software modernization can be achieved: missing entries in the event log indicate that process parts implemented in the legacy software system have never been used. This information is vital for modernizing legacy software systems as the code fragments may correspond to technical debt and must therefore not be migrated [11].

5.2 Recording User Interactions

Once the code tracker is installed, we are able to document the interactions of users with the legacy application, including resulting software interactions. For recording user interactions, we support two variants [12]:

Silent Recording shall record the use of the legacy application, starting with the login a of user until closing the legacy application. We allow specifying which information shall be recorded and in which form. For example, personal data may only be logged in an anonymized way. By only recording selected user sessions (e.g., sessions of users from a certain department), we can further restrict the recording of user interactions to relevant user groups (e.g., users handling invoices) in a fine-grained fashion.

Dedicated Recording aims to record existing (i.e., already identified) processes implemented in the legacy software system. Users may define the start and end of the recording (e.g., through predefined key combinations), and provide additional information about the recorded process. This, in turn, allows for a precise delimitation of the interactions corresponding to a process.

6 Evaluation

The evaluation of our approach is threefold: First, we assess whether the identified requirements are met. Second, we analyze the performance of an Oracle legacy software system to which we applied our approach. Third, we applied process discovery algorithms to the derived event logs and evaluate the resulting process models with domain experts. In total, the legacy software system used to evaluate the approach comprises 589 database tables with 9977 columns. Additionally, 60712 database statements (including more than 8000 different statements) were implemented in a total of over 5 million lines of code. Furthermore, the legacy software system comprises 1285 forms and 6243 different screens. The event log was created using dedicated recording (cf. Section 5). In other words, the users in this event log were able to provide additional information of the recorded business process (e.g., name and description of the process). Additionally, we applied the approach using silent recording to the legacy software system of an insurance company[1].

6.1 Requirements Evaluation

To evaluate *Requirement 1 (Relevance)*, according to which the event log shall solely contain process-relevant information, we conduct an in-depth and automatic analysis of the legacy software system by identifying and clustering important tables and source code fragments (cf. Section 4). This enables us to distinguish between relevant and non-relevant information. As a result, we are able to configure the code tracker to ensure that only relevant data is collected.

Requirement 2 (Scope) deals with the scope of the legacy software system and aims to minimize the amount of domain knowledge needed for the analysis. By analyzing the source code, we are able to identify which code fragments refer to which database tables. Clustering the database tables (cf. Section 4) allows grouping the tables that belong to the same context. This enables a best guess approach that may be checked by domain experts to further improve the event log generation. Compared to alternative approaches (e.g., extensive interviews), our approach requires significantly less domain knowledge.

Requirement 3 (Consistency) refers to consistency with respect to data types, timestamps, and resources. While we account for consistency regarding data types (e.g., timestamp formats and variables), due to the automated nature of our approach, the fulfillment of this requirement also depends on the consistency of the analyzed legacy software system as well as the underlying database.

According to *Requirement 4 (Performance)* the event log generation must not affect the performance of the legacy software system or user interactions with the legacy application. Typically, the generation of redo log files, archive log files based on the redo log files, as well as the log rotation capabilities are tuned to not influence the performance of the analyzed legacy software system. For further analysis, the generated event log is extracted asynchronously to ensure that the extraction neither impacts users nor the performance of the running legacy software system. Additionally, the logging of user interactions focuses on the relevant actions identified during legacy software analysis. Furthermore, the logging is running in a separate, isolated transaction to the user session.

[1] Event logs provided: https://cloudstore.uni-ulm.de/s/7jYeRnXtcsk2Wfd

Finally, the collected event data is also persisted in a separate storage to not affect performance.

6.2 Performance Analysis

To further evaluate the performance effects of our approach on the considered legacy software system, we executed the same 3 processes multiple times (N=10) with and without event log generation and measured the duration of the following performance metrics: navigation, loading time, and function call. Note that due to limitations of the legacy software system, timestamps could only be collected every 10 milliseconds. In other words, differences of up to 20 ms might exist. Figs. 5 - 7 depict the collected performance metrics. When navigating through the legacy software system the average duration decreased by 25 ms. The average loading times decreased by 30 ms after adding the event log generation. These differences are in range of the timestamp limitations of the legacy software system. Therefore, we can conclude that the event log generation does not significantly impact navigation and loading times. On average, the duration of function calls increased by 0.65 seconds (+18.2%) per function call. However, after closer inspection, this increase is mainly due to recursive function calls that generate event log entries with each iteration. We are able to only record one event log entry for recursive function calls, consequently reducing the increase to the level of non-recursive function calls. Concerning the latter, we observed an average increase of 14 ms (1.73%). Across all observed performance metrics, the differences do not impact typical user and software interactions.

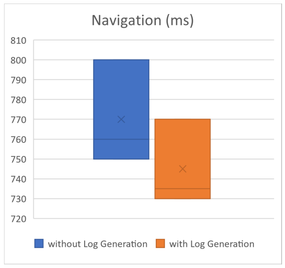


Fig. 5: Navigation

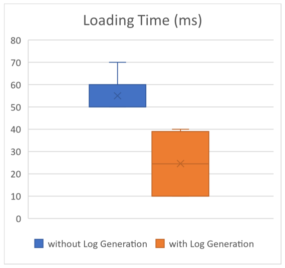


Fig. 6: Loading Time

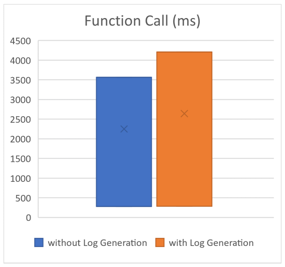


Fig. 7: Function Calls

6.3 Initial Process Discovery

We applied several process discovery algorithms to the event logs generated with our approach using default algorithm configurations. Next, we showed the resulting process models to domain experts (N=13) and asked them to evaluate to which degree they are able to recognize the legacy software system in each process model on a 5-Point Lickert scale from *not at all* to *completely*. Overall, the domain experts rated the process model generated by the Heuristic Miner (threshold = 0.9) best (Mean = 4.45, SD = 0.63). This indicates that process models discovered from the generated event log adequately represent the behavior of processes implemented by the legacy software system.

Table 1: Domain Expert Recognition of Discovered Process Models (N=13)

	Inductive (Tree)	Inductive (BPMN)	DFG	Heuristic (thold=0.75)	Heuristic (thold=0.9)	Heuristic (thold=0.95)
Mean (SD)	3.08 (1.07)	3.08 (0.73)	2.31 (1.2)	4.15 (0.77)	4.46 (0.63)	3.38 (1.27)

While the results could be improved using additional process discovery algorithms or fine-tuning parameters, they emphasize the high quality of generated event logs as no additional event log preparation was required.

7 Related Work

This paper is related to event log generation, robotic process automation, and legacy software system analysis.

Process mining algorithms require event logs and, therefore, the generation of event logs from various sources has gained great attention [13]. Databases are often used as the main resource for extracting event data from information systems [10, 14]. A quality-aware and semi-automated approach to extract event logs from relational data is presented in [15]: users may select event log attributes from available data columns, assisted by data quality metrics. In the context of legacy software systems, however, relying solely on the information present in databases is not sufficient, as important process-relevant knowledge is often captured in the source code as well as the displayed user forms, but cannot be discovered from the database solely. For example, legacy databases are often not normalized and miss important information, e.g., foreign key constraints.

In the field of Robotic Process Automation (RPA) [16], user interface interactions and software robots are used to replicate human tasks. An approach for recording the interactions with user interfaces and the generation of user interface event logs is presented in [17]. A pipeline of processing steps enabling robotic process mining tools to generate RPA scripts from UI logs is presented in [18]. [19] presents an UI logger that generates an event log from multiple user interfaces. As opposed to [17–19], our approach accounts for the effects on the legacy software system (e.g., exact database statements), i.e., it does not only consider the user interface interactions in isolation.

In [20], a framework to recover workflows from an e-commerce scenario is presented, leveraging static analysis to identify business knowledge from source code. Similarly, [21] presents an approach for recovering business knowledge from legacy application databases by inspecting the data stored within the database. As our approach also aims to identify business knowledge from legacy software systems, it differs from [20, 21]. Instead of extracting business knowledge from static analysis, we generate event logs that represent business knowledge using interactions with the legacy software systems.

[22] deals with the generation of event logs from legacy software systems by first extending the source code and then recording the event logs. In contrast, our approach requires less domain knowledge for generating the event logs as we derive relevant source code fragments from the clusters identified in the

database (including foreign-key constraints specified in the source code) rather than domain experts or system analysts. Additionally, we support two event log generation variants (silent and dedicated) that enable further insights into specific processes implemented in the legacy software system.

8 Summary and Outlook

This paper presented an approach for generating event logs from running legacy software systems with minimal domain knowledge. We combine information from source code analysis and the database structure to identify tables and source code fragments relevant in the context of supported business processes.

Further, we identify which database tables and source code fragments may correspond to a specific process (e.g., handling an invoice) using a cluster analysis. We then automatically inject event log generation functions to the legacy software system to track user and software interactions with the legacy software system, while at the same time recording the resulting database transactions. Next, we document user interactions with the application and the resulting database changes from the running legacy application in a user-decided fashion. We then combine both logs to correlate user interactions with corresponding database changes to obtain event logs suitable for process mining.

We evaluated the approach based on the requirements identified with domain experts, a performance analysis of the legacy software system, and the application and evaluation of initial process discovery algorithms. The requirements are met, enabling the generation of comprehensive event logs from legacy software systems with the approach. A performance evaluation using an Oracle legacy software system has shown that our event log generation does not impact the performance of the legacy software system, and initial process models discovered were able to adequately represent the legacy software system for domain experts using the event logs generated with the approach.

In future work, we will apply the presented approach to additional legacy software systems. Additionally, we will increase the quality of the discovered process models for non-experts using more intuitive event log labels based on the legacy software system.

Acknowledgments This work is part of the SoftProc project, funded by the KMU-innovativ Program of the Federal Ministry of Education and Research, Germany (F.No. 01IS20027A)

References

1. W. M. P. van der Aalst, A. Adriansyah, A. K. A. De Medeiros, F. Arcieri, T. Baier, T. Blickle, J. C. Bose, P. Van Den Brand, R. Brandtjen, J. Buijs *et al.*, "Process mining manifesto," in *Int'l Conf on BPM.* Springer, 2011, pp. 169–194.
2. A. Rozinat and W. M. P. van der Aalst, "Conformance checking of processes based on monitoring real behavior," *Information Systems*, vol. 33, no. 1, pp. 64–95, 2008.

3. M. Dumas, M. L. Rosa, J. Mendling, and H. A. Reijers, *Fundamentals of Business Process Management*, 2nd ed. Springer, 2018.
4. M. Feathers, *Working effectively with legacy code.* Addison-Wesley, 2013.
5. W. M. P. van der Aalst, *Process Mining: Data Science in Action*, 2nd ed. Springer Berlin Heidelberg, 2016.
6. R. J. Wieringa, *Design science methodology for information systems and software engineering.* Springer, 2014.
7. L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: Bringing order to the web.” Stanford InfoLab, Technical Report 1999-66, 1999, previous number = SIDL-WP-1999-0120.
8. B. Fluri, M. Wursch, M. Pinzger, and H. Gall, “Change distilling:tree differencing for fine-grained source code change extraction,” *IEEE Transact' on Softw Eng*, vol. 33, no. 11, pp. 725–743, 2007.
9. E. G. L. de Murillas, W. M. P. van der Aalst, and H. A. Reijers, “Process mining on databases: Unearthing historical data from redo logs,” in *Business Process Management.* Springer, 2015, pp. 367–385.
10. W. M. P. van der Aalst, *Extracting Event Data from Databases to Unleash Process Mining.* Springer, 2015, pp. 105–128.
11. W. Cunningham, “The wycash portfolio management system,” *SIGPLAN OOPS Mess.*, vol. 4, no. 2, 1992.
12. M. Breitmayer, L. Arnold, and M. Reichert, “Towards retrograde process analysis in running legacy applications,” in *Proceedings of the 14th ZEUS Workshop*, vol. 3113. CEUR-WS.org, 2022, pp. 11–15.
13. D. Dakic, D. Stefanovic, T. Lolic, D. Narandzic, and N. Simeunovic, “Event log extraction for the purpose of process mining: A systematic literature review,” in *Innov' in Sust' Mngmt and Entr'.* Springer, 2020, pp. 299–312.
14. D. Calvanese, M. Montali, A. Syamsiyah, and W. M. P. van der Aalst, “Ontology-driven extraction of event logs from relational databases,” in *BPM Workshops.* Springer, 2016, pp. 140–153.
15. R. Andrews, C. van Dun, M. Wynn, W. Kratsch, M. Röglinger, and A. ter Hofstede, “Quality-informed semi-automated event log generation for process mining,” *Decision Support Systems*, vol. 132, p. 113265, 2020.
16. J. Wewerka and M. Reichert, “Robotic process automation - a systematic mapping study and classification framework,” *Enterprise Information Systems*, 2022.
17. D. Choi, H. R'bigui, and C. Cho, “Enabling the gab between rpa and process mining: User interface interactions recorder,” *IEEE Access*, vol. 10, pp. 39 604–39 612, 2022.
18. V. Leno, A. Polyvyanyy, M. Dumas, M. La Rosa, and F. Maggi, “Robotic process mining: Vision and challenges,” *Bus' & Inf' Sys' Eng'*, vol. 63, 06 2021.
19. J. M. López-Carnicer, C. del Valle, and J. G. Enríquez, “Towards an open-source logger for the analysis of rpa projects,” in *Business Process Management: Blockchain and Robotic Process Automation Forum.* Springer, 2020, pp. 176–184.
20. Y. Zou and M. Hung, “An approach for extracting workflows from e-commerce applications,” in *14th IEEE ICPC'06*, 2006, pp. 127–136.
21. R. Pérez-Castillo, D. Caivano, and M. Piattini, “Ontology-based similarity applied to business process clustering,” *J. Softw. Evol. Process*, vol. 26, no. 12, pp. 1128–1149, 2014.
22. R. Pérez-Castillo, B. Weber, J. Pinggera, S. Zugal, I. G. R. de Guzmán, and M. Piattini, “Generating event logs from non-process-aware systems enabling business process mining,” *Enterp. Inf. Syst.*, vol. 5, no. 3, pp. 301–335, 2011.