Process Mining in Software Systems

Discovering Real-Life Business Transactions and Process Models from Distributed Systems

Maikel Leemans (m.leemans@tue.nl) and Wil M. P. van der Aalst (w.m.p.v.d.aalst@tue.nl)
Eindhoven University of Technology, P.O. Box 513, 5600 MB, Eindhoven, The Netherlands.

***Abstract*—This paper presents a novel reverse engineering technique for obtaining real-life event logs from distributed systems. This allows us to analyze the operational processes of software systems under real-life conditions, and use process mining techniques to obtain precise and formal models. Hence, the work can be positioned in-between reverse engineering and process mining.**

We present a formal definition, implementation and an instrumentation strategy based the joinpoint-pointcut model. Two case studies are used to evaluate our approach. These concrete examples demonstrate the feasibility and usefulness of our approach.

***Index Terms*—Reverse Engineering, Process Mining, Distributed Systems, Event Log, Process Discovery, Performance Analysis, Joinpoint-Pointcut Model, Aspect-Oriented Programming**

I. Introduction

A. Behavior Analysis of Real-Life Processes

System comprehension, analysis and evolution are largely based on information regarding the structure, behavior and operation of the System Under Study (SUS). When no complete information regarding the behavior is available, one has to extract this information through dynamic analysis techniques.

Typically, dynamic behavior is captured in the form of *process models*. These models describe the dynamic, operational and interactive aspects of the SUS. This dynamic behavior is often an emergent product of the *intercommunication* between the different *system components*. The resulting behavior of these communicating components is often not well understood, which makes process models particularly interesting. In addition to understanding the dynamic behavior, better insight into the operational aspects, including monitoring real-life performance, is critical for the success of software systems.

Besides monitoring real-life behavior, there is a need to support dynamic analysis with precise and formal models. *Unified Modeling Language (UML) diagrams* have become the de facto standard for describing software. However, UML diagrams have no precise semantics, are not directly usable for model-based techniques, and do not support performance analysis. In contrast, *event logs* show the actual behavior and, hence, serve as a starting point for process mining. The combination of event logs and process mining techniques provides a powerful way to discover formal process models and analyze operational processes based on event data.

In this paper, *we define a novel reverse engineering technique for obtaining real-life event logs from distributed software systems, spanning across multiple system components*. This allows us to analyze the operational processes of software systems under real-life conditions, and use process mining techniques to obtain precise and formal models (see for example Figures 4 and 5). Unlike conventional approaches (e.g. profilers), our approach provides an integrated view, across system components, and across perspectives (performance, end-to-end control flow, etc.).

B. On Reverse Engineering Dynamic Models

Any dynamic analysis approach based on reverse engineering techniques must address the following concerns:

Information Retrieval Strategy This concern addresses *how* information is obtained from the SUS. One has to choose a retrieval technique (e.g., an *instrumentation strategy*), for which constraints on the *target language* have to be considered. In addition to how information is obtained, one has to address *which* information is to be obtained, and at which level of detail (i.e., the *granularity*). Finally, one has to take into account the *environment* that actually triggers behavior in the SUS.

Information Collecting and Processing Strategy This concern addresses how information obtained from the SUS is *collected and processed*. First of all, in a distributed context, one has to specify a *data collecting infrastructure* to combine information streams from the different system components. Next, one has to specify a *target model*. On the one hand, there is the question of *correlating information*, especially in the context of a distributed SUS (e.g., which events belong together). On the other hand, there are *inter-component* and *inter-thread* aspects to be considered, as well as *timing issues* in a distributed system.

Analysis Strategy This concern addresses how, using the obtained information and target model, the SUS is actually *analyzed*. One has to consider how to interpret the resulting artifacts, and how analysis questions can be answered. This analysis ranges from *discovering* control-flow patterns to *performance analysis*; and ranges from finding *deviations* with respect to expected behavior to finding *anti-patterns*.

C. Overview of Methodology

Our high-level strategy for analyzing distributed systems consists of the following steps (see also Figure 1):

1) We instrument the SUS code or binary with tracing code using instrumentation advices (see Section III-E).
2) We gather event data from real-life user requests, and convert the data into event logs by discovering business transactions (see Section III-F).
3) With the resulting event log we can answer various analysis questions using process mining techniques.

Note that our methodology does not require detailed input about the SUS. In fact, many details about the SUS are discovered from data present in the generated event log. We only need some predicates (pointcuts, see Section III-E) specifying areas of interest in the SUS. With these pointcuts, we automatically add tracing code that generates event data upon execution.

D. Goal and Assumptions

Our goal is to analyze the operational processes of software systems. In particular, we target at analyzing the functional

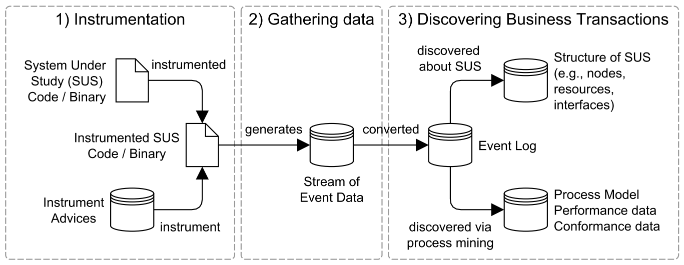

1) Instrumentation
2) Gathering data
3) Discovering Business Transactions
System Under Study (SUS) Code / Binary
instrumented
Instrumented SUS Code / Binary
Instrument Advices
instrument
generates
Stream of Event Data
converted
Event Log
discovered about SUS
Structure of SUS (e.g., nodes, resources, interfaces)
discovered via process mining
Process Model Performance data Conformance data

Fig. 1. Overview of our high-level strategy and the corresponding input and output. See Section III for a detailed discussion of the terminology used.

perspective, focusing on user requests. The reverse engineering technique we use to achieve this goal is designed for any instrumentable programming language. That is, our approach and results are language-independent.

Our approach supports the analysis of distributed systems, provided they rely on point-to-point communication. We will abstract from the details of local clocks in a distributed setting, as addressed in Subsection IV-B.

The current realization is restricted to a single thread per node. However, the introduced definitions can be extended to support multi-threaded distributed software systems, as will be discussed in Section VI.

E. Outline

The remainder of this paper is organized as follows. Section II positions the work in existing literature. A detailed definition of the System Under Study (SUS) as well as the conversion from system events to an event log is given in Section III. The novel notion of discovering business transactions, as well as our information retrieval and processing strategies are discussed in Section IV. The approach and implementation are evaluated in Section V using two case studies, performed on existing open-source software. Section VI concludes the paper.

II. Related Work

Reverse engineering models from systems is not new, and there exists a lot of literature on extracting models from (software) systems. In the area of dynamic systems analysis, there exist techniques targeted at understanding the *behavior* of systems. In this section we start with a comparison of various reverse engineering techniques, discussing the current trends and their advantages and disadvantages in Subsection II-A. We conclude this section by discussing the techniques available in the area of process mining in Subsection II-B.

A. Dynamic Analysis Techniques

Overview of Dynamic Analysis Techniques: Dynamic system analysis techniques are used for understanding the *behavior* of systems. Understanding the dynamic part of a system is often not possible by just analyzing the source code, especially in the face of polymorphism and dynamic binding. An even greater challenge is discovering the interactive aspect of the system, which is an emergent product of the *intercommunication* between the different *system components*. There is an abundance of literature with proposals and techniques addressing dynamic analysis techniques. In Table I we have compared several approaches targeted at non-distributed and distributed systems, evaluated using the following criteria:

Distributed *Whether the technique is designed for a distributed or non-distributed setting.*

Granularity *The level of detail of the analysis technique.* In [1], [3], [4], [5], [6], the behavior is captured down to the control-flow level (i.e., down to the loop and if-else statements). At the other end, in [7], [9], [10], [11] the behavior is captured at the high-level components level (i.e., external interfaces). In [2], only the end-user "pages" are captured (i.e., the user interface level).

Information Retrieval Strategy *The techniques used for retrieving dynamic information.* Lion's share of the existing techniques uses some form of instrumentation (either source code transformation and/or binary weaving) to add tracing code to the SUS [1], [2], [3], [6], [8], or adapt existing tracing code [11]. In [1], source code analysis is used for enriching the dynamic information. An altogether different technique is used in [7], where network packages are monitored, outside of the SUS.

Environment *The environment that triggers behavior in the system.* Most of the techniques only considered discovering the control-flow aspect of the system behavior, and thus used black-box testing techniques for triggering the system [1], [2], [3], [4], [5], [6], [10], [11]. A few techniques also looked at the behavior in a real-life environment [7], [8], [9].

Target Language *Restrictions on the target language for which the approach is defined.* Frequently, the instrumentation tool AspectJ is used (see [12]), thereby targeted at the Java programming language [1], [6], [8], [11]. In addition, for relating events, [6] assumes the Java RMI distributed middleware, [9] the COBRA distributed middleware, and [7] assumes only low-level TCP/IP communication.

Distributed Events Correlation *How, in a distributed setting, events across different components are correlated.* Not many techniques explicitly addressed the concern of correlating events across components in a distributed setting. In [6], correlation is handled by introducing extra communication with correlation indicators. The authors of [7] relied on deep packet inspection, retrieving sender and receiver information. The process of inspecting communication channels used in [10] is similar to our technique.

Target Model *The type of target model produced by the approach.* Lion's share of the techniques produces a UML Sequence Diagram (UML SD) [1], [2], [6], [7]. The authors of [11] produce Communicating Finite State Machines (CFSM). In [8], the authors specified a set of events, called a Monitor Log, as target model.

Shortcomings of Current Approaches: The majority of the techniques considered relies on a testing environment, and produce a UML Sequence Diagram. In addition, in many cases the issue of correlating distributed events is not addressed explicitly.

One complication is that the produced UML models are imprecise; they have no precise semantics, are not directly usable for model-based techniques, and do not support performance analysis. Several proposals in literature attempted to address this by defining a precise subset for UML [13], [14], or translating UML models into precise models like Petri nets [15], [16]. However, in the translation steps from events via abstractions like UML to Petri net models, valuable information is lost. A better approach would be to go directly from events to precise models with clear semantics, thus enabling the use

TABLE I
STRATEGY COMPARISON OF DYNAMIC ANALYSIS TECHNIQUES

	Author	Distributed	Granularity	Information Retrieval Strategy	Environment	Target Language	Correlation of Distributed Events	Target Model
[1]	Labiche	-	Control-flow	Instrumentation + source	Testing	Java via AspectJ	n/a	UML SD
[2]	Alalfi	-	"User pages"	Instrumentation	Testing	Scripting (PHP) via TXL	n/a	UML SD
[3]	Briand	-	Control-flow	Instrumentation	Testing	C++	n/a	UML SD
[4]	Oechsle	-	Control-flow	Java debug interface	Testing	Java	n/a	UML SD
[5]	Systä	-	Control-flow	Customized debugger	Testing	Java	n/a	UML SD-like
[6]	Briand	+	Control-flow	Instrumentation	Testing	Java + RMI via AspectJ	Extra communication	UML SD
[7]	Ackermann	+	Components	Monitor network packets	Real-life	TCP/IP	Network packet from/to	UML SD
[8]	Van Hoorn	+	Varied	Instrumentation	Real-life	Java via AspectJ	No correlation	Monitor Log
[9]	Moe	+	Components	Call interceptors	Real-life	COBRA	No correlation	Performance statistics
[10]	Salah	+	Components	JVM profiler	Testing	Java	Comm. channel from/to	UML SD
[11]	Beschastnikh	+	Components	Given log, Instrument.	Testing	Log-only, Java via AspectJ	Predef. comm. channels	CFSM
	Leemans M.	+	Interfaces	Instrumentation	Real-life	Instrumentable + TCP/IP	Comm. channel from/to	Process models, with, i.a., performance info

of model-based analysis and techniques.

Related to the issues of imprecise UML models and the use of model-based techniques is the lack of insight into the performance aspect of system behavior [16]. As observed by the authors of [8], real-live monitoring may enable early detection of quality-of-service problems, and may deliver usage data for resource management. Therefore, we argue that we need to discover *precise models reflecting real-life behavior*.

B. Event Logs and Process Mining

In order to obtain process models, we rely on event logs. An event log can be viewed as a multiset of *traces* [17]. Each trace describes the life-cycle of a particular *case* (i.e., a *process instance*) in terms of the *activities* executed. In Subsection III-D a formal definition for event logs is given, and in [18], [19], corresponding meta-model, implementations and standardized exchange format are defined.

Process mining techniques use event logs to discover, monitor and improve real-life processes [17]. The three main process mining tasks are:

Process discovery: Learning a process model from example behavior recorded in an event log.

Conformance checking: Aligning an event log and a process model for detecting and diagnosing deviations between observed (logged) behavior and modelled behavior.

Performance analysis: Replaying observed behavior on process models for identifying bottlenecks, delays and inefficiencies in processes.

Many process discovery techniques have been presented in literature. These techniques produce precise models and are readily available through the Process Mining Toolkit ProM [19]. A variety of discovery techniques yielding Petri nets [20], [21], [22], [23] and process trees [24], [25] were proposed. By aligning an event log and a process model, it is possible to perform advanced conformance and performance analysis [26]. Current state of the art techniques also looks into leveraging additional information like *location* (which is also present in our event log) to produce more accurate models [27]. In addition, additional insights can be gained through investigating organizational information (e.g., resource collaboration) [28] and partial order information [29].

III. DEFINITIONS

Before we can discuss the different strategies we developed, we need a clear understanding of the System Under Study (SUS) and event logs. We start out with some basic preliminaries in Subsection III-A. Next, we present our view on distributed systems in Subsection III-B. After that, we quickly cover the basic principles of process mining (Subsection III-C) and event logs (Subsection III-D). Finally, we will discuss the basic principle used for instrumenting the SUS (Subsection III-E), and the conversion from system events to an event log (Subsection III-F). The key concepts and their relations are captured in the domain model shown in Figure 2.

A. Preliminaries

Sequences: Sequences are used to represent traces in an event log.

Given a set X, a sequence over X of length n is denoted as $\sigma = \langle a_1, a_2, \ldots, a_n \rangle \in X^*$. We denote the empty sequence as $\langle\rangle$.

Intervals: Intervals are used to define the start and end time of an event.

Let $I = \{\, (i, j) \mid i \leq j \,\} \subset \mathbb{N}^2$ be the set of intervals. We use $\perp \notin I$ to denote an invalid (empty) interval. Given an interval $x = (i, j) \in I$, we write $x_s = i$ and $x_e = j$ for the start and end of an interval respectively.

We define the following relations on I, with $x, y \in I$:

$$x = y \overset{\text{def}}{=} ((x_s = y_s) \wedge (x_e = y_e)) \qquad \text{equality}$$

$$x \sqsubseteq y \overset{\text{def}}{=} (y_s \leq x_s \leq x_e \leq y_e) \qquad \text{containment}$$

$$x \sqsubset y \overset{\text{def}}{=} ((x \sqsubseteq y) \wedge (x \neq y)) \qquad \text{strict containment}$$

$$x \cap y \overset{\text{def}}{=} \begin{cases} z, & \text{if } x \neq \perp \wedge y \neq \perp \wedge z \in I; \\ \perp, & \text{otherwise.} \end{cases} \qquad \text{intersection}$$

$$\textbf{with } z = (\max(x_s, y_s), \min(x_e, y_e))$$

$$x \cup y \overset{\text{def}}{=} \begin{cases} z, & \text{if } x \neq \perp \wedge y \neq \perp; \\ \perp, & \text{otherwise.} \end{cases} \qquad \text{union}$$

$$\textbf{with } z = (\min(x_s, y_s), \max(x_e, y_e))$$

B. The System Under Study: Anatomy of a Distributed System

In this section, we present our view on distributed systems. To make things more concrete, we will map our view onto an imaginary distributed software system.

A *distributed system* consists of a set of interacting *system components*, distributed over a set of *logical platforms*. Each system component is instantiated on a *node*, and can offer *services* via its *external interfaces*. Each logical platform can deploy multiple nodes, that is, multiple instantiations of system components. In our imaginary software system, our system components could be a business and a data component: a webserver and database, respectively. The logical platforms

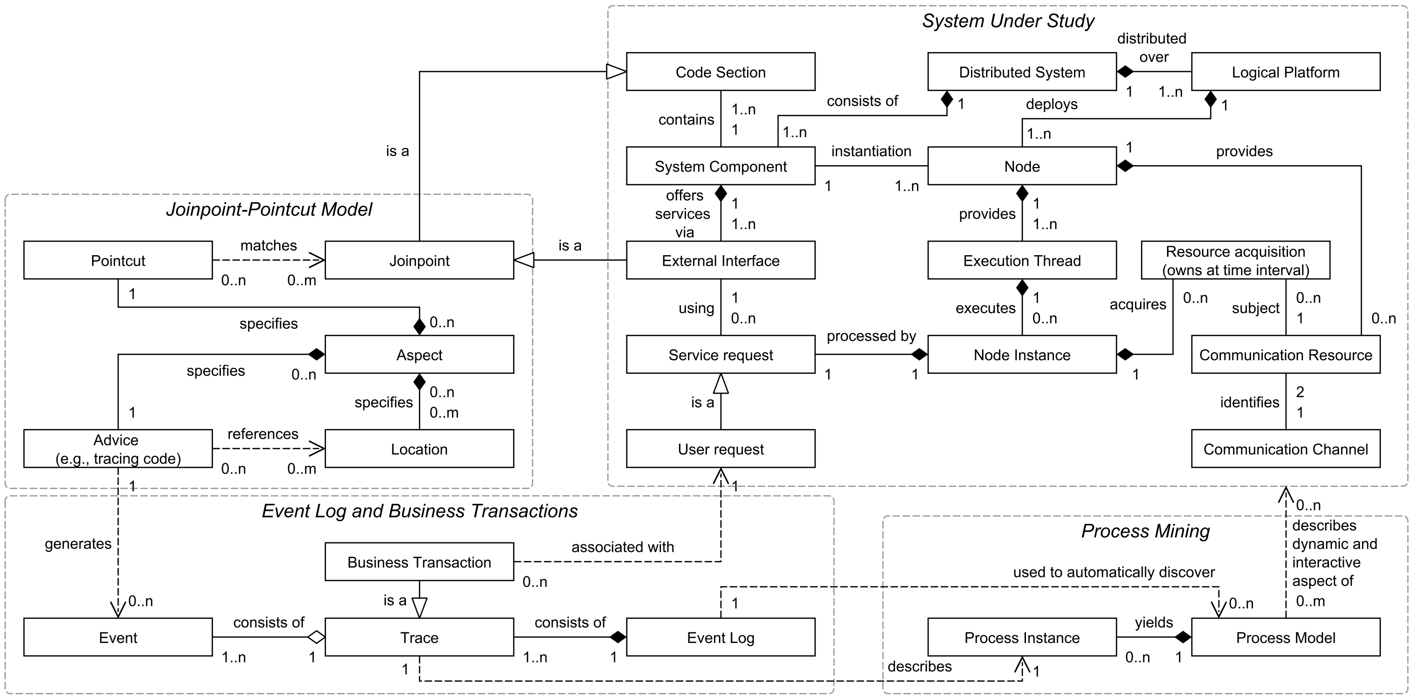


Fig. 2. Domain model illustrating the key concepts and their relations (using UML notation). See Section III for a detailed description of each concept.

would be the servers in the datacenter. The nodes are instances of the webserver and database. Note that the webserver and database may be instantiated multiple times, for example, in a load-balancing setting. In the latter case, it would make sense to host the resulting nodes on different servers (i.e., different logical platforms). The webserver offers webpage services, while the database offers query services.

A *service request*, denoting the use of an external interface, is processed on a *node* by a *node instance*. We assume that a *node instance* is executed by one of the *execution threads* provided by a node, but future work can remove this restriction. A service can be requested by a node part of the SUS (i.e., *intercommunication*), or by an external user (i.e., a *user request*). In our imaginary software system, the webserver (the node) would have a pool of execution threads. Whenever the client browser requests a webpage service (a user request), the webserver assigns one of the available threads to process this request. For this user request, the node instance is executed by the assigned thread.

Whenever two nodes are communicating to execute a service request, they are sharing a *communication channel*. A communication channel is identified by a pair of *communication resources*, each representing one *endpoint* of the communication channel. Node instances can *acquire* communication resources provided by the node in order to communicate. The moment a communication resource is used by a node instance, that node instance *owns* that resource. Note that communication resources can be reused, and thus can have different owners at different moments in time. See also the group labeled "System Under Study" in Figure 2. In our imaginary software system, a communication channel could be a TCP/IP connection. An endpoint is the combination of an IP address and a port on a node. Hence, a communication resource is a pair of 'from' and 'to' endpoints.

For the rest of the paper, we use the formal definition below for referencing these concepts.

Definition 1 (Nodes and Communication Resources): We denote the set of nodes with N, the set of node instances at node $n \in N$ with T_n, and let $T = \bigcup_{n \in N} T_n$. Furthermore, let R_n denote the set of communication resources available at node $n \in N$, and let $R = \bigcup_{n \in N} R_n$. In addition, let $r \sim r'$ denote that the two communication resources $r \in R_n, r' \in R_{n'}$, with $n, n' \in N \wedge n \neq n'$ identify the same communication channel. We impose the following constraint on the above defined sets:

$$\forall n, n' \in N,\ n \neq n' : (R_n \cap R_{n'}) = (T_n \cap T_{n'}) = \emptyset$$

C. Process Mining

Recall, in Subsection II-B we covered how process mining techniques use event logs to discover, monitor and improve real-life processes. Each trace in the event log describes the life-cycle of a particular *process instance*. Together, these process instances describe the behavior of the SUS.

As a passive learning technique, the quality of the resulting model depends on the quality and volume of the behavior that has been observed. It is therefore important to obtain a large, high quality event log in order to build an accurate model. See also the group labeled "Process Mining" in Figure 2.

D. Event Log and Business Transactions

Event Log: The starting point for any process mining technique is an *event log*, which is formally defined below.

Definition 2 (Event, Trace, and Event Log): Let E_L be the set of *events* occurring in the event log. A *trace* is a sequence $\sigma \in E_L^*$ of events. An *event log* $L \subseteq E_L^*$ is a collection of traces. Each trace corresponds to an execution of a process, i.e., a *case* or *process instance*.

Business Transactions: In the context of analyzing system behavior, we recognize a special type of traces, called *business transactions*. A business transaction consists of a sequence of related events, which together contribute to serve a *user request*.

Events in a single case can span multiple nodes. Recall, a user request is a service requested by an *external* user. Hence, a business transaction captures the emergent product of the inter-communication between system components required for one external interface exposed by the software system as a whole.

See also the group labeled "Event Log and Business Transactions" in Figure 2.

E. Joinpoint-Pointcut Model

In order to obtain an event log detailing the dynamic behavior of the SUS, we instrument the SUS with tracing code to generate the necessary events. This tracing instrumentation should minimize the impact on the SUS, and provide as little human overhead as possible. Note that the behavior of the instrumented SUS may be different from the unmodified SUS, especially in the context of deadlines. This is an unavoidable consequence; observing a system changes the system [30]. However, we should nevertheless strive to minimize the impact of the tracing instrumentation.

To make the instrumentation less intrusive and as systematic as possible, we use the *joinpoint-pointcut model* frequently used in the area of Aspect-Oriented Programming (AOP) [31]. This way, developers can work on the clean, unmodified code, and we can monitor *any* SUS that can be instrumented *without manually modifying the source code*.

A *joinpoint* is a point in a running program where additional behavior can be usefully joined or added. A joinpoint needs to be addressable and understandable by an ordinary programmer to be useful. A *pointcut* is an expression (predicate) that determines whether a given joinpoint matches. An *advice* is a piece of code (e.g., event trace code) to be added. An *aspect* defines an advice to be added at all joinpoints matched by a specified pointcut. Hence, an aspect is a container detailing how to instrument what. Each tracing aspect typically instruments a particular (possible user-defined) *location* in a software system, such as a method in a particular component, library or layer. See also the group labeled "Joinpoint-Pointcut Model" in Figure 2. In Section IV-A we will use the joinpoint-pointcut model to define which parts of a software system we wish to include in our event log.

In the remainder of the paper, we use the formal definition below for referencing these concepts.

Definition 3 (Joinpoint-Pointcut Model): Let $J_n \subseteq \mathcal{U}_J$ denote the set of joinpoints available at node $n \in N$. A pointcut is a predicate matching a subset of joinpoints. In addition, we denote the set of locations with $\mathcal{L}$. We impose the following constraint on the above defined set:

$$\forall n, n' \in N,\ n \neq n' : (J_n \cap J_{n'}) = \emptyset$$

F. From System Events to Event Logs

System Events: Recall, we are interested in discovering business transactions: a sequence of related events, which together contribute to serve a *user request*. Therefore, we specify the structure of *system events* (described below), capturing enough information to relate events *within* and *across the external interfaces* of system components. Using the data in these system events, we can discover business transactions, the basis for instantiating our event log.

Definition 4 (System Events): Let E be the *set of system events*, such that every $e \in E$ has the following structure:

$$e = (i : I,\ n : N,\ t : T_n,\ j : J_n,\ r : R_n \cup \{\bot\},\ l : \mathcal{L})$$

We write $e.n$ to access attribute n in event e.
The interpretation of the above attributes is as follows:

- $i : I$ models the time interval of the event. That is, it models the call (start) and return (end) of the involved joinpoint. Typically, this will correspond to the entry and exit of a method, see also Section IV-A.
- $n : N$ models the node on which the event was generated.
- $t : T_n$ models the node instance, which generated the event.
- $j : J_n$ models the joinpoint that was executed to generate this event.
- $r : R_n \cup \{\bot\}$ models the (optional) communication resource associated with this event.
- $l : \mathcal{L}$ models the location specified in the aspect that instrumented this joinpoint.

Communication and Related Events: In order to obtain business transactions, we need to correctly *cluster* events. To correctly cluster events, we will use the notion of *communication intervals* and *related events*.

Definition 5 (Communication intervals): Given a resource $r \in R$, we can get the set of events E_r that are associated with r:

$$E_r \overset{\text{def}}{=} \{\, e \in E \mid e.r = r \,\}$$

Recall, a node instance *owns* a resource during the time it uses a resource, and a resource can have different owners at different moments in time. To define the time interval where a node instance owns a resource $r \in R$, we look for 'evidence' in the form of events associated with resource r. Given two events $e, e' \in E_r$ associated with resource $r \in R_n$. If the two events have the same node instance $t \in T_n$, and there is no other event $e'' \in E_r, e''.t \neq t$ in between e and e', then we know that during the time $e.i \cup e.i'$, node instance t owns resource r. Formally, the set of intervals where $t \in T_n$ owns $r \in R_n$ is defined by function h_R:

$$\begin{aligned} h_R(t, r) \overset{\text{def}}{=} \{\, & i = (e.i \cup e'.i) \mid e, e' \in E_r,\ e.t = e'.t = t, \\ & \neg(\exists e'' \in E_r,\ e''.t \neq t : (e''.i \cap i) \neq \bot) \,\} \end{aligned}$$

Given the definition above, we can define the set of maximal intervals where $t \in T_n$ owns $r \in R_n$ (i.e., the *communication intervals*) as follows:

$$f_R(t, r) \overset{\text{def}}{=} \{\, i \in h_R(t, r) \mid \neg(\exists i' \in h_R(t, r) : i \sqsubset i') \,\}$$

Definition 6 (Related events): Two events $x, y \in E, x \neq y$ are directly related, notation $x \rightarrow y$, iff either:

1) x and y are part of the same node instance, and $x.i$ is contained in $y.i$.
2) y started before x, and there exists related resources $r_x, r_y \in R,\ r_x \sim r_y$ that are at a certain point in time owned by $x.t$ and $y.t$ respectively (see Definition 5).

Formally, the directly related relation is defined as follows:

$$\begin{aligned} (x \rightarrow y) \overset{\text{def}}{=} (\ & (x \neq y) \wedge (&& \text{Distinct.} \\ & (\ x.t = y.t \wedge x.i \sqsubseteq y.i\ ) && \text{Case 1.} \\ & \vee (\ x.t \neq y.t \wedge y.i_s \leq x.i_s \wedge && \text{Case 2.} \\ & \quad \exists r_x, r_y \in R,\ r_x \sim r_y : \\ & \quad \exists i_x \in f_R(x.t, r_x),\ i_y \in f_R(y.t, r_y) : \\ & \quad ((x.i \cap i_x) \cap (y.i \cap i_y)) \neq \bot \\ & \quad)\,)) \end{aligned}$$

Intuitively, an event x is directly related to y (i.e., $x \rightarrow y$) if x is 'caused' by y (e.g., in case 1, called by). Note that $\rightarrow$ is an irreflexive and antisymmetric relation.

Two events $x, y \in E, x \neq y$ are related, notation $x \twoheadrightarrow y$, iff there is a path from x to y in the relation $\rightarrow$. Formally:

$$\begin{aligned} (x \twoheadrightarrow y) \overset{\text{def}}{=} &((\, x \rightarrow y) && \text{Base case.} \\ &\vee (\, \exists z \in E : x \twoheadrightarrow z \wedge z \rightarrow y)) && \text{Step case.} \end{aligned}$$

Note that $\twoheadrightarrow$ is an irreflexive, antisymmetric and transitive relation.

Let $X \in \mathcal{P}(E)$ be the set of subsets of related events, and let $Y \subseteq X$ be the set of maximal subsets of related events (i.e., the basis for business transactions). Formally:

$$X \overset{\text{def}}{=} \{\, \{e\} \cup \{\, e' \in E \mid e' \twoheadrightarrow e \,\} \mid e \in E \,\}$$

$$Y \overset{\text{def}}{=} \{\, x' \in X \mid \neg(\exists x'' \in X : x' \subset x'') \,\}$$

Instantiating Event Logs: Using the set Y of maximal subsets of related events, we can now instantiate an event log L. We construct our event log $L \subseteq E_L^*$ from the set of log-events E_L by finding all valid business transactions. The set of log-events E_L is obtained from the set of events E by mapping each event to a start and end event, based on its interval.

Definition 7 (Event Log Instantiation): For an event $e \in E$, the set of log-events $f_L(e)$ corresponds to the start and end of $e.i$. The set E_L of log-events is the union of all mapped events. Formally:

$$E_L \overset{\text{def}}{=} \bigcup_{e \in E} f_L(e) \textbf{ with } f_L(e) \overset{\text{def}}{=} \{(e.i_s, e),\ (e.i_e, e)\}$$

The event log $L \subseteq E_L^*$ (see Definition 2) of business transactions is based on Y (see Definition 6), and defined as:

$$L \overset{\text{def}}{=} \left\{ \sigma \in E_L^* \;\middle|\; \mathit{sorted}(\sigma) \wedge \exists x' \in Y : \{e \in \sigma\} = \bigcup_{e \in x'} f_L(e) \right\}$$

Note that we assumed a total ordering on the elements of E_L, where events are sorted by time. In the edge case that two related events are logged with the same timestamp (i.e., a tie), in the resulting trace σ, the tie is handled as per a stable sort algorithm.

IV. Methodology and Realization

Using the definition from Section III, we will now discuss the different strategies we developed. We start out with detailing our information retrieval strategy in Subsection IV-A, specifying how we instrument the System Under Study (SUS) with tracing code. After that, we will cover the collecting and information processing strategy in Subsection IV-B, specifying how we gather the event data and how we convert it into an event log. As a mental reference, please consult Figure 3 as a sample application of our strategies.

A. Dynamic Information Retrieval Strategy

Recall from Subsection III-E, that we use the joinpoint-pointcut model to instrument the SUS with tracing code. This makes the instrumentation less intrusive and as systematic as possible. In addition, instrumentation minimizes the impact of the tracing instrumentation on the SUS and on the developers. In this section we will discuss how we use the joinpoint-pointcut model as the basis for our information retrieval strategy. Note that the question of 'what information has to be logged' is answered by our definition of system events, as discussed in Subsection III-F.

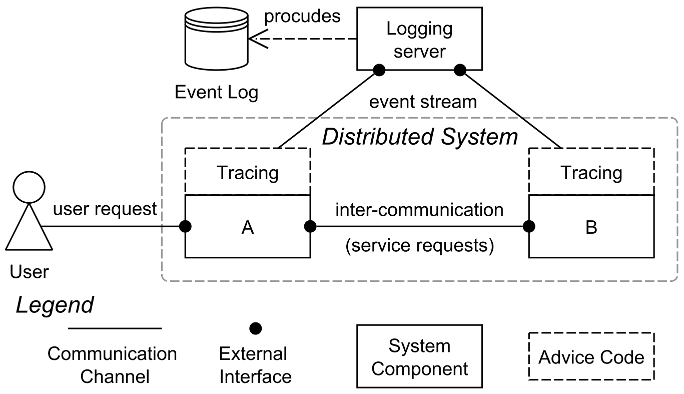


Fig. 3. Domain infrastructure model illustrating the implementation of our information retrieval, collecting and information processing strategy.

Instrumentation Strategy: Our instrumentation strategy focuses on the resulting behavior from the intercommunication between different system components via their external interfaces. Therefore, we recognize two types of joinpoints we will target at: *communication resource joinpoints* and *interface joinpoints*. The tracing advice code added via pointcuts targeting these types of joinpoints adds invocations to a logging client interface.[1] This logging client interface generates events and streams these to a logging server; see also Figure 3.

The communication resource joinpoint type is about recognizing when a node instance has acquired which communication resource. This type of joinpoint can be matched by application-independent language-dependent pointcuts targeting low-level network communication methods. The advice code could, for example, recover the involved communication resource based on the socket involved. Note that this type of aspect needs to be defined only once for any programming language.

The interface joinpoint type is about recognizing (external) interfaces, thus providing a context for the system analyst. This type of joinpoint can be matched by application-dependent method pointcuts defined by the system analyst. The advice code associated with interface joinpoint aspects is application-independent language-dependent. This means that the system analyst only has to define the pointcuts and possible location information, and the rest of the aspect is handled in a generic, automated fashion. Note that this type of advice code needs to be defined only once for any programming language.

Granularity of Information Retrieval: Our instrumentation aspects are designed to capture enough information to relate events *within* and *across the external interfaces* of system components. Hence, we are primarily retrieving information on the system component instances. Secondarily, through the option to specify interface pointcuts, we can provide more detailed context information.

The advice code added to the SUS records event data on the method level. The moments when a method is entered and exited correspond to the start and end of the event time interval. Each event is enriched with the current node and joinpoint information, available through the pointcut specification. The current node instance information is a piece of application-independent language-dependent information. In our evaluation, we used the notion of thread ids for instance identification

[1]Java tracing advice code online: https://svn.win.tue.nl/repos/prom/XPort/

within a node. Note that node instance identification needs be defined only once for any programming language.

This way, all the information specified for system events in Subsection III-F is accounted for.

Environment: Most analysis techniques assume some controlled environment in which the SUS is executed. Frequently, through the use of black-box testing techniques, many scenarios (i.e., user requests) are triggered. However, our approach focuses on capturing dynamic information about real-life usage of the SUS. Therefore, instead of relying on testing techniques for generating executions, we assume real-life interaction. One concern may be that this approach cannot cover enough relevant distinct scenarios. But since we want to understand the real-life usage of the system, it stands to reason that the most relevant scenarios are those scenarios that are (frequently) triggered by the actual users of the system.

Target Language Considerations: Any language for which method-level AOP techniques exist can be supported after defining the basic communication resource and interface advices. For the Java language one can use techniques like Java Agents and Javassist [32], [33] or AspectJ [12]. The C++ language can be supported via the use of, for example, AspectC++ [34] or AOP++ [35]. Most other languages can be supported via source transformation techniques like the TXL processor [36].

B. Information Collecting and Processing Strategy

In Subsection IV-A, we have discussed how we get our SUS to generate events, and stream these to a logging server; see also Figure 3. In this section we will detail how this information is processed into actual event logs. In addition, we will touch on two related issues: 1) event timing in a distributed context, and 2) communication within system components.

Collecting Strategy: The events generated by the logging clients are streamed to one central logging server. Although not required by the basic definitions from Section III, for practical reasons we assume an offline event processing strategy. Therefore, our collecting strategy on the server side will be a simple merge of event streams, and storing the results on disk. Future work could focus on enabling the processing strategy for real-time event stream processing, but we did not consider this for our initial proof of concept.

Event Processing Strategy: The event processing strategy consists of an algorithm that takes the set of generated events, and produces an event log. That is, we cluster events into traces based on the notion of related events (see Subsection III-F). Our basic algorithm consists of two parts: 1) discovering ownership of communication resources, and 2) discovering actual traces.

The discovery of communication resources ownership is essentially the function $f_R(t, r)$ in Definition 5. We simply traverse the recorded events in ascending order and build the resource acquisition intervals.

The discovery of actual traces is now possible by resolving the 'related events' mapping. For each pair of events $x, y \in E,\ x \neq y$ we can calculate $x \rightarrow y$ by checking the two cases of the formal definition in Definition 6. For finding resources r_x, r_y and corresponding acquisition intervals i_x, i_y, we can use the function $f_R(t, r)$. In order to find r_x, we can simply investigate the domain of $f_R(x.t)$.

In order to obtain the actual event log, we use the idea presented in Definitions 6 and 7. We consider each event $e \in E$ for which there is no $e' \in E$ such that $e \twoheadrightarrow e'$, and create a trace out of all the events $e'' \in E$ with $e'' \twoheadrightarrow e$.

Timing Issues: Until now we have assumed that time across platforms in a distributed system can be used for ordering and comparing event intervals. However, in a distributed system, local clocks (i.e. the time on different platforms) can be different. To remedy this issue, we assume the Network Time Protocol (NTP) is used to synchronize platform clocks via the internet (see also RFC 5905 [37]). For systems not connected to the internet, similar algorithms can be employed to synchronize with a local global clock server.

Inter-Thread Communication: Recall from Subsection III-B, we assume that a *node instance* is executed by *one* of the *execution threads* provided by a node. This restricts the current implementation to a single thread per node instance. Future work can remove this restriction, and we will elaborate on this in Section VI.

V. Evaluation

This section discusses two case studies using existing open-source software to demonstrate the feasibility and usefulness of our approach. In addition, the instrumentation overhead is investigated via a performance benchmark.

A. Methodology

We used two existing open-source software applications for our experiments. The first is a pet catalog demo available in the NetBeans IDE [38]. The second is Wordcount MapReduce example available in the Apache Hadoop distributed computing framework (version 2.6.0) [39], [40]. For these experiments we used a laptop with a 2.40 GHz CPU, Windows 8.1 and Java SE 1.7.0 67 (64 bit) with 4 GB of allocated RAM. The instrumentation is done via Java Agents, using Javassist 3.19.0-GA [32], [33].

For the two case studies we indicate the instrumentation pointcuts used (i.e., the input). After instrumentation, we simulate a batch of user requests, collect the data and process it to obtain event logs. Finally, we will use process discovery and performance analysis to answer the following analysis questions:

1) What is the high-level end-to-end process corresponding to the interfaces of the System Under Study?
2) What are the main bottlenecks or areas for improvements in this process?

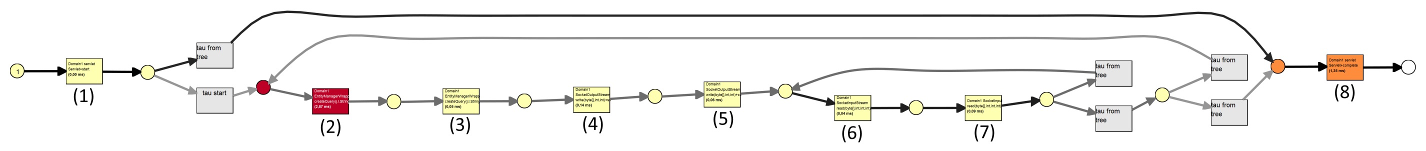
B. Case Study - Pet Catalog

The pet catalog software system consists of a JavaEE webserver implementation. At the front external interface, users can issue webpage requests, handled via JavaEE servlets. At the backend, the software interfaces with a MySQL database through the JavaEE persistence interface.

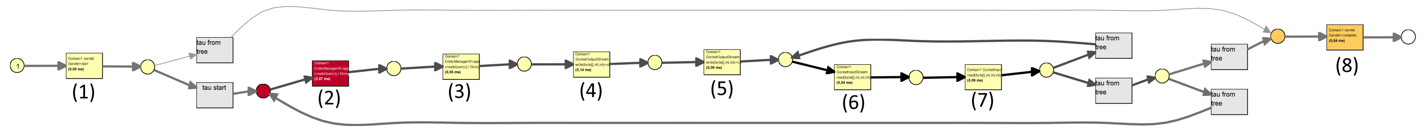
Instrumentation Pointcuts: To target communication resource joinpoints, we defined pointcuts targeting Java socket read and writes, as well as the JavaEE servlet interface. To target interface joinpoints, we defined pointcuts targeting the JavaEE persistence interface. The exact pointcuts are:

```
HasInterface: javax.persistence.EntityManager
Communication: java.net.SocketInputStream,
java.net.SocketOutputStream, javax.servlet.*,
javax.faces.*
```

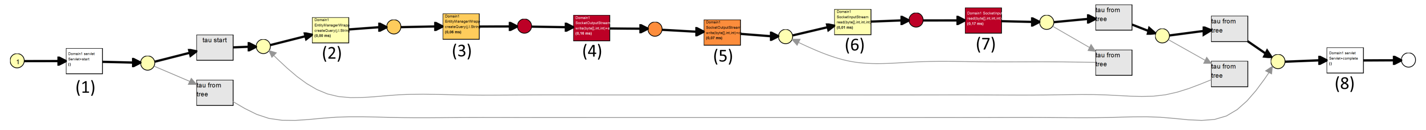
Note that this, together with the actual SUS, is only the input we need.



(a) Using the complete event log for performance analysis

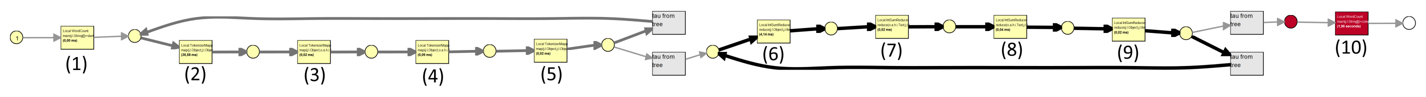


(b) Using a filtered event log (filtered after extraction) for performance analysis, focusing only on cases with database queries

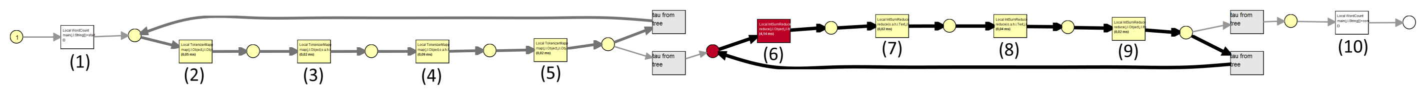


(c) Using a filtered event log (filtered after extraction) for performance analysis, exclusively focusing only on the database query events

Fig. 4. Process model of the Pet Catalog software system, depicted as a Petri net overlaid with performance information as per the replay algorithm [26].
Transitions (rectangles) from left to right read: (1) `Servlet+start`, (2) `EntityManagerWrapper.createQuery()+start`, (3) `EntityManagerWrapper.createQuery()+complete`, (4) `SocketOuputStream.write()+start`, (5) `SocketOuputStream.write()+complete`, (6) `SocketInputStream.read()+start`, (7) `SocketInputStream.read()+complete`, (8) `Servlet+complete`.



(a) Using the complete event log for performance analysis



(b) Using a filtered event log (filtered after extraction) for performance analysis, ignoring the wrapping main() method

Fig. 5. Process model of the Wordcount MapReduce job on the Apache Hadoop software system, depicted as a Petri net overlaid with performance information as per the replay algorithm [26].
Transitions (rectangles) from left to right read: (1) `WordCount.main()+start`, (2) `TokenizerMapper.map()+start`, (3) `TokenizerMapper.map()+start`, (4) `TokenizerMapper.map()+complete`, (5) `TokenizerMapper.map()+complete`, (6) `IntSumReduce.reduce()+start`, (7) `IntSumReduce.reduce()+start`, (8) `IntSumReduce.reduce()+complete`, (9) `IntSumReduce.reduce()+complete`, (10) `WordCount.main()+complete`. (Outer map() reduce() methods are untyped variants, the inner methods are typed variants.)

High-level End-to-end Process: The process model displayed in Figure 4 was discovered via the inductive miner [25]. Each transition (rectangle) represents the start or end (i.e., the call and return, respectively) of a method. The beginning (left) and end (right) of this model is the call and return of the Servlet front external interface. In between are the activities during a Servlet call.

At the beginning of executing the Servlet, there is a choice to perform some query requests or skip querying the database. In the case the decision is to query the database, then a query statement is first created (i.e., prepared). After that, there is some communication with the database to issue the request and receive the results (write and read). When done receiving results, we have the option to loop back and perform additional queries, or to finish the Servlet process.

Main Bottlenecks: The color highlighting applied in Figure 4 is the result of replaying (a filtered version of) the event log on the process model [26]. The coloring on the transitions and places indicate waiting time between calls, between returns or between a start and end (i.e., a call and return, and hence, throughput). Dark red and light yellow colors indicate a high or low waiting/throughput time, respectively. The color scale is automatically computed by the replaying algorithm. The arc coloring indicate how frequently that branch is used (over all traces), with black and gray denoting a high and low frequency respectively.

As can be seen near the choice in the beginning in Figure 4(a), in most cases querying the database is skipped. The average case throughput, from start to end, is 2,77 milliseconds.

By applying the appropriate filter to our event log, we obtain the in Figure 4(b), showing only the cases with database interaction. Note that no re-instrumentation was needed, process mining techniques provide advanced filtering after extraction [17], [19]. The average case throughput with this log, from start to end, is 7,96 milliseconds. There are two large delays visible, before and after the query loop, indicated by the red circle before (2) and the orange circle before (8). These delays correspond with the Servlet startup and shutdown (in this case, a JavaServer Facelet).

After filtering out the Servlet start and complete events, we obtain the Figure 4(c), focusing only on the database querying. In this filtered view, thanks to a rescaled performance color scale, we see a delay between creating queries (the orange circle between (2) and (3)) and the actual database communication (the red circle after (3)). In addition, a similar delay is visible during reading results from the database (the red circle before (7)).

Conclusion: After specifying a few simple pointcuts, the end-to-end process was quickly discovered. For these pointcuts, no real in-depth knowledge or manual coding was needed, allowing a quick instrumentation of the system. Through the use of performance analysis, the Servlet startup and shutdown, as well as the transition between query preparation and communication were identified as the main bottlenecks. Since in most cases the database is not queried (based on frequency information), the latter bottleneck could be considered less of an issue.

C. Case Study - Wordcount MapReduce

The Wordcount MapReduce job is a simple application that counts the number of occurrences of each word in a given input set [40]. The job is executed on the Hadoop MapReduce framework. We used the English version of the Universal Declaration of Human Rights as input [41]. The front-end of the application is the MapReduce client, whose main function sets up the Wordcount job. The back-end of the application, i.e., the actual map() and reduce() tasks, are executed on a different (local) node. In between these nodes is the Hadoop MapReduce system.

Instrumentation Pointcuts: To target communication resource joinpoints, we defined pointcuts targeting Java channel socket read and writes. To target interface joinpoints, we defined pointcuts targeting the client main and backend map and reduce interfaces. The exact pointcuts are:

```
Interfaces: org.apache.hadoop.examples.*.map(*,
org.apache.hadoop.examples.*.reduce(*,
org.apache.hadoop.examples.*.main(*
Communication: java.net.SocketInputStream,
java.net.SocketOutputStream,
java.nio.channels.SocketChannel
```

Note that this, together with the actual SUS, is only the input we need.

High-level End-to-end Process: The process model displayed in Figure 5 was discovered via the inductive miner [25]. The beginning (left) and end (right) of this model is the call and return of the main() method. The remained of the model occurs inside, or during this main() method.

The are two main phases in the MapReduce job: the first loop is for the map() methods, and the second loop is for the reduce() methods. Note that both map() and reduce() consists of two functions: the generic interface implementation, and the typed variant that is called during executing the generic method.

Main Bottlenecks: Again, we replayed (a filtered version of) the event log on the process model, resulting in Figure 5.

As can be seen in Figure 5(a), the methods in the second phase (i.e., the reduce methods) are executed more frequently: roughly 4 times the frequency of the map methods. Taking into account that the wordcount job computes the number of occurrences of each word, we conclude that there are many small reduce tasks as a result of a few map tasks. The average case throughput for this application, from start to end, is 17,96 seconds. The biggest delays are at the beginning and end of the job, before the first map call, and after the final reduce call.

After filtering out the main() start and complete, we obtain the Figure 5(b), focusing only on the map and reduce methods. In this filtered view, thanks to a rescaled color scale, we discover a delay between map-reduce and reduce-reduce transitions.

Conclusion: Again, a few simple pointcuts and no real in-depth knowledge or manual coding sufficed to quickly discover the end-to-end process. Although the system under study is rather complex, the initial effort needed to start analyzing its behavior is very small. Through the use of performance analysis, the MapReduce job startup and shutdown, as well as map-reduce and reduce-reduce transitions were identified as the main bottlenecks.

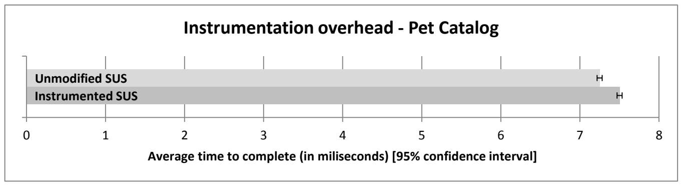
D. On Instrumentation Overhead

We investigate the extend of instrumentation overhead via a performance benchmark. We measure the time to complete of both the Pet Catalog and the Wordcount MapReduce software for both the instrumented and unmodified version of the software. This time measurement is performed on the user side, and thus includes the communication overhead between the SUS and the client user. By measuring on the user side, we can measure the time of the unmodified software, and influence the SUS as little as possible. We repeated these measurements several times, and calculated the average runtime and associated 95% confidence interval. The results are presented in Figure 6.

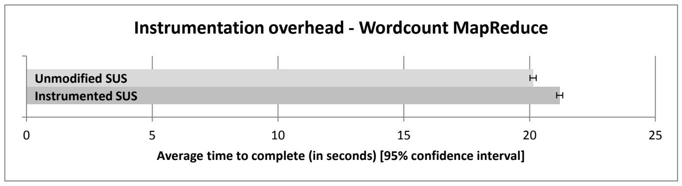
For the Pet Catalog case, we requested a batch of 5000 webpages involving database querying. As shown in Figure 6(a), the difference in performance is very small.

For the Wordcount MapReduce case, we performed a batch of 60 jobs in sequence. As shown in Figure 6(b), we see a small difference in runtime.

Although in both cases the difference is observable, it is very small compared to the total time to complete. Based on the above presented observations, we conclude that the impact of the instrumentation is negligible.



(a) The Pet Catalog case



(b) The Wordcount MapReduce case

Fig. 6. Effect of instrumentation on the average time to complete.

VI. CONCLUSION

In this paper, we presented a novel reverse engineering technique for obtaining real-life event logs from distributed systems. This allows us to analyze the operational processes of software systems under real-life conditions, and use process mining techniques to obtain precise and formal models. In addition, process mining techniques can be used to monitor and improve processes via, for example, performance analysis and conformance checking. We presented a formal definition, implementation and an instrumentation strategy based on the joinpoint-pointcut model.

Two case studies demonstrated how, after specifying a few simple pointcuts, we quickly gained insight into the end-to-end process, and the main performance bottlenecks in the context of this process. By changing the pointcut specifications, and filtering the obtained event logs, system analysts can easily select the right amount of detail. Through the use of frequency and performance information we can determine the seriousness of discovered bottlenecks.

The current implementation is limited to single-threaded distributed software systems, but future work will look into adapting Definition 6 (related events) to handle multi-threaded software. Note that, in essence, the related events relation in Definition 6 relates threads based on inter-thread communication. That is, this basic definition does not assume that the correlated threads are on different nodes, and could also be applied in the context of inter-thread communication within one node. When adapting this definition, one should pay special attention to inter-thread communication via data-structures, such as present in typical producer-consumer settings.

In this paper, we assumed an offline process analysis setting, but developing process mining techniques supporting event streams could yield valuable real-time insight. Finally, with the advent of software event logs, which are rich in data and semantics, new process mining techniques could focus on making location information and subprocesses explicit in the discovered models.

References

[1] Y. Labiche, B. Kolbah, and H. Mehrfard, "Combining static and dynamic analyses to reverse-engineer scenario diagrams," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, Sept 2013, pp. 130–139.

[2] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "Automated reverse engineering of UML sequence diagrams for dynamic web applications," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*. IEEE, April 2009, pp. 287–294.

[3] L. C. Briand, Y. Labiche, and Y. Miao, "Towards the reverse engineering of UML sequence diagrams," *2013 20th Working Conference on Reverse Engineering (WCRE)*, p. 57, 2003.

[4] "Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface (jdi)," in *Software Visualization*, ser. Lecture Notes in Computer Science, S. Diehl, Ed., 2002, vol. 2269.

[5] T. Systä, K. Koskimies, and H. Müller, "Shimba – an environment for reverse engineering java software systems," *Software: Practice and Experience*, vol. 31, no. 4.

[6] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the reverse engineering of UML sequence diagrams for distributed java software," *Software Engineering, IEEE Transactions on*, vol. 32, no. 9, pp. 642–663, Sept 2006.

[7] C. Ackermann, M. Lindvall, and R. Cleaveland, "Recovering views of inter-system interaction behaviors," in *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*. IEEE, Oct 2009, pp. 53–61.

[8] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst, "Continuous monitoring of software services: Design and application of the Kieker framework," Kiel University, Research Report, November 2009. [Online]. Available: http://eprints.uni-kiel.de/14459/

[9] J. Moe and D. A. Carr, "Using execution trace data to improve distributed systems," *Software: Practice and Experience*, vol. 32, no. 9.

[10] M. Salah and S. Mancoridis, "Toward an environment for comprehending distributed systems," *2013 20th Working Conference on Reverse Engineering (WCRE)*, p. 238, 2003.

[11] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014, pp. 468–479.

[12] J. D. Gradecki and N. Lesiecki, *Mastering AspectJ. Aspect-Oriented Programming in Java*. John Wiley & Sons, 2003, vol. 456.

[13] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting, "A subset of precise UML for model-based testing," in *Proceedings of the 3rd International Workshop on Advances in Model-based Testing*, ser. A-MOST '07, 2007, pp. 95–104.

[14] "Defining precise semantics for UML," in *Object-Oriented Technology*, ser. Lecture Notes in Computer Science, G. Goos, J. Hartmanis, J. van Leeuwen, J. Malenfant, S. Moisan, and A. Moreira, Eds., 2000, vol. 1964.

[15] S. Bernardi, S. Donatelli, and J. Merseguer, "From UML sequence diagrams and statecharts to analysable petri net models," in *Proceedings of the 3rd International Workshop on Software and Performance*, ser. WOSP '02, 2002, pp. 35–45.

[16] J. P. López-Grao, J. Merseguer, and J. Campos, "From UML activity diagrams to stochastic Petri nets: Application to software performance engineering," in *Proceedings of the 4th International Workshop on Software and Performance*, ser. WOSP '04, 2004, pp. 25–36.

[17] W. M. P. van der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer-Verlag, Berlin, 2011.

[18] C. W. Günther and H. M. W. Verbeek, "XES – standard definition," 2014. [Online]. Available: http://repository.tue.nl/777826

[19] "XES, XESame, and ProM 6," in *Information Systems Evolution*, ser. Lecture Notes in Business Information Processing, P. Soffer and E. Proper, Eds., 2011, vol. 72.

[20] W. M. P. van der Aalst, A. J. M. M. Weijters, and L. Maruster, "Workflow Mining: Discovering Process Models from Event Logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 9, pp. 1128–1142, 2004.

[21] W. M. P. van der Aalst, A. K. A. de Medeiros, and A. J. M. M. Weijters, "Genetic Process Mining," in *Applications and Theory of Petri Nets 2005*, ser. Lecture Notes in Computer Science, G. Ciardo and P. Darondeau, Eds. Springer-Verlag, Berlin, 2005, vol. 3536, pp. 48–69.

[22] R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser, "Process Mining Based on Regions of Languages," in *International Conference on Business Process Management (BPM 2007)*, ser. Lecture Notes in Computer Science, G. Alonso, P. Dadam, and M. Rosemann, Eds., vol. 4714. Springer-Verlag, Berlin, 2007, pp. 375–383.

[23] W. M. P. van der Aalst, V. Rubin, H. M. W. Verbeek, B. F. van Dongen, E. Kindler, and C. W. Günther, "Process mining: a two-step approach to balance between underfitting and overfitting," *Software & Systems Modeling*, vol. 9, no. 1, pp. 87–111, 2010.

[24] "On the role of fitness, precision, generalization and simplicity in process discovery," in *On the Move to Meaningful Internet Systems: OTM 2012*, ser. Lecture Notes in Computer Science, R. Meersman, H. Panetto, T. Dillon, S. Rinderle-Ma, P. Dadam, X. Zhou, S. Pearson, A. Ferscha, S. Bergamaschi, and I. Cruz, Eds., 2012, vol. 7565.

[25] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Discovering Block-structured Process Models from Incomplete Event Logs," in *Applications and Theory of Petri Nets 2014*, ser. Lecture Notes in Computer Science, G. Ciardo and E. Kindler, Eds., vol. 8489. Springer-Verlag, Berlin, 2014, pp. 91–110.

[26] A. Adriansyah, "Aligning observed and modeled behavior," Ph.D. dissertation, Technische Universiteit Eindhoven, 2014.

[27] W. M. P. van der Aalst, K. A., V. Rubin, and H. M. W. Verbeek, "Process discovery using localized events," 2015, to appear in Petri nets in 2015.

[28] M. Song and W. M. P. van der Aalst, "Towards comprehensive support for organizational mining," vol. 46, no. 1. Springer-Verlag, Berlin, 2008, pp. 300–317.

[29] M. Leemans and W. M. P. van der Aalst, "Discovery of frequent episodes in event logs," in *Proceedings of the 4th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2014)*. CEUR-ws.org, 2014.

[30] W. Schutz, "On the testability of distributed real-time systems," in *Reliable Distributed Systems, 1991. Proceedings., Tenth Symposium on*. IEEE, Sep 1991, pp. 52–61.

[31] T. Elrad, R. E. Filman, and A. Bader, "Aspect-oriented programming: Introduction," *Communications of the ACM*, vol. 44, no. 10, pp. 29–32, Oct. 2001.

[32] S. Chiba, "Javassist – a reflection-based programming wizard for java," in *Proceedings of OOPSLA98 Workshop on Reflective Programming in C++ and Java*, October 1998, p. 5.

[33] ——, "Load-time structural reflection in java," in *European Conference on Object-Oriented Programming 2000 – Object-Oriented Programming*, ser. Lecture Notes in Computer Science, E. Bertino, Ed. Springer Berlin Heidelberg, 2000, vol. 1850, pp. 313–336. [Online]. Available: http://dx.doi.org/10.1007/3-540-45102-1_16

[34] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An aspect-oriented extension to the C++ programming language," in *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, ser. CRPIT '02. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2002, pp. 53–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=564092.564100

[35] "AOP++: A generic aspect-oriented programming framework in C++," in *Generative Programming and Component Engineering*, ser. Lecture Notes in Computer Science, R. Glck and M. Lowry, Eds., 2005, vol. 3676.

[36] J. R. Cordy, "The TXL source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190 – 210, 2006, special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA 04). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642306000669

[37] D. Mills, J. Martin, J. Burbank, and W. Kasch, "Network time protocol version 4: Protocol and algorithms specification," *IETF RFC5905*, June 2010. [Online]. Available: http://tools.ietf.org/html/rfc5905

[38] NetBeans, "Pet catalog - Java EE 6 sample application," https://netbeans.org/kb/samples/pet-catalog.html, [Online, accessed 17 April 2015].

[39] The Apache Software Foundation, "Apache hadoop," http://hadoop.apache.org/, [Online, accessed 17 April 2015].

[40] ——, "Mapreduce tutorial," https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html, [Online, accessed 17 April 2015].

[41] United Nations, "Universal declaration of human rights," http://www.un.org/en/documents/udhr/, [Online, accessed 17 April 2015].