# Discovery of Functional Architectures From Event Logs

Jan Martijn E.M. van der Werf and Erwin Kaats

Department of Information and Computing Science
Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
j.m.e.m.vanderwerf@uu.nl, e.j.kaats@students.uu.nl

**Abstract.** The functional architecture focuses on decomposing functionality into modules that offer certain features. These features require interactions in order to complete their functionality. However, functional architectures typically only focus on the static aspects of the system design. Additional modeling techniques, such as message sequence charts are often used in the early phases of software design to indicate how the software should behave.
In this paper we investigate the use of process discovery techniques to discover from these scenarios the internal behavior of individual components. Based on event logs, this paper presents an approach (1) to derive the information flows between features, (2) identify the internal behavior of features, and (3) to discover the order between features within a module. The approach results in a sound workflow model for each module. We illustrate the approach using a running example of a payment system.

## 1 Introduction

One of the principle tasks of a software architect is to design a software system [17], i.e., to organize the software elements the system is composed of in sets of structures, to allow reasoning about the system [4]. Many different Architectural Description Languages (ADLs) exist to document software architecture. However, due to the large competitive market in the software product industry, architecture is often neglected in software product organizations [14]. Hence, not many ADLs are used in practice. As experienced in [14], in software product organizations, architects rather use informal architectural models as an instrument of communication and discussion.

An important aspect of software architecture is the functionality it offers. To decompose and specify the functionality of software, the authors of [6] introduced the Functional Architecture Model (FAM), which offers the desired modeling technique used by many software architects in software product organizations [14]. The FAM separates the functionality into so-called features that are offered by the different modules the system is decomposed into.

Features interact with other features via information flows to offer their functionality. However, FAM only offers a static view on this interaction, i.e., the information flow only shows possible interactions, but imposes no order on or dependencies between these flows. Thus, to show how functionality is offered by the system, the architect requires additional models. One way is to define scenarios on top of the models, in which the architect can specify which features interact in which order. These scenario then result in event logs, that can be analyzed using process mining techniques [2]. Another source for discovering the possible interactions between features is the use of system execution data [21], mapping events to the (partial) execution of features. In this way, execution data can be used to reconstruct a software architecture.

In software product organizations, time to market is often a more important priority than having a properly documented software architecture. Consequently, architecture documentation is often outdated or even missing [9]. Therefore, discovering architectural models help such organizations in maintaining their software products. In this paper, we investigate the possibility to use process mining techniques to discover, the functional architecture of the system from an event log. We thereby focus on three basic questions on the functional architecture:

1. Which features interact?
2. What is the internal behavior of features?
3. What is the order in which features are executed within a module?

The first question focuses on the discovery of information flows: given an event log, is it possible to derive which features interact? Next, we investigate whether it is possible to derive the internal behavior of features based on event logs. In other words, we focus on the question how does a feature use its information flows to complete its functionality. The last question deals with the high-level view of the functional architecture. To execute the system's functionality, the features within a module are called in a certain order. Can process discovery techniques be used to discover these orders?

The remainder of this paper is structured as follows. To illustrate the approach, Sec. 2 presents a running example which we will use throughout the paper. Next, Sec. 3 presents the basic notions used in the paper. Section 4 introduces the functional architecture model in more detail, after which in Sec. 5 we will focus on solving the three questions posed in the introduction. Section 6 concludes the paper.

## 2 Running Example

As an running example, consider the Payment System as introduced in [10]. The system consists of three modules, *Debtor*, *Payment* and *Creditor*. The payment module serves as an intermediate between the Debtor and the Creditor. An example of such a payment module is the european SEPA standard. The payment module initiates a transaction, which the debtor needs to accept. If the debtor accepts, the payment is continued, and the creditor is contacted to start the

transaction. If for some reason the creditor rejects the transaction, the debtor is notified, and the transaction is terminated. Similarly, if the creditor accepts, the payment is passed to the debtor, and finally, the creditor receives the final payment information.

As the software evolved into the current system, no precise model exists that specifies the behavior of this system. The system only recorded the order in which the different features of the modules have been called in an event log, as shown in Tbl. 1. Each pair in the table represents the feature and the module to which that feature belongs. For readability, the features and modules are abbreviated in this event log.

The system is decomposed into three modules: the *Debtor* module (X), the *Payment* (Y) module, and the *Creditor* (Z). Based on the event log, the software architect finds the following features:

- Receive transaction request (A);
- Reject transaction (B);
- Accept transaction (C);
- Cancel transaction (D);
- Initiate payment (E);
- Send payment details (F);
- Archive transaction request (G).
- Send transaction request (H);
- Reject transaction request (I);
- Initiate creditor (J);
- Cancel transaction (K);
- Initiate payment (O);
- Handle payment (M);
- Archive transaction (N).
- Start transaction (Q);
- Handle transaction (S).

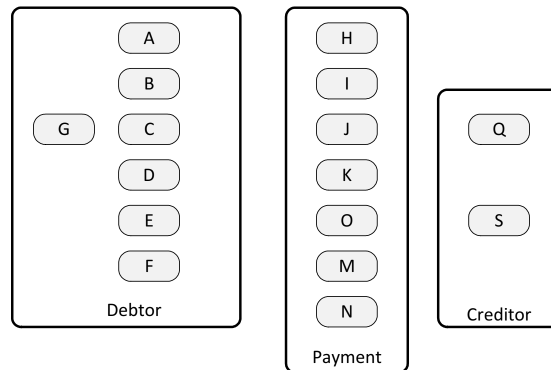



**Fig. 1.** Initial functional architecture model of the running example

| Case | Trace |
|---|---|
| 1 | (H, Y), (A, X), (B, X), (G, X), (I, Y), (N, Y) |
| 2 | (H, Y), (A, X), (B, X), (I, Y), (G, X), N, Y) |
| 3 | (H, Y), (A, X), (B, X), (I, Y), (N, Y), (G, X) |
| 4 | (H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (K, Y), (D, X), (G, X), (N, Y) |
| 5 | (H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (K, Y), (D, X), (N, Y), (G, X) |
| 6 | (H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (K, Y), (N, Y), (D, X), (G, X) |
| 7 | (H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z),( O, Y), (E, X), (F, X), (G, X), (M, Y), (S, Z), (N, Y) |
| 8 | (H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (G, X), (M, Y), (N, Y), (S, Z) |
| 9 | (H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (G, X), (S, Z), (N, Y) |
| 10 | (H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (G, X), (N, Y), (S, Y) |
| 11 | (H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (S, Z), (G, X), (N, Y) |
| 12 | (H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (S, Z), (N, Y), (G, X) |
| 13 | (H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (N, Y), (G, X), (S, Z) |
| 14 | (H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (N, Y), (S, Y), (G, X) |

**Table 1.** System execution data of the payment system

Based on this information, the architect can draw the modules with their features, as shown in Fig. 1. In the remainder of this paper, we investigate a method to use event logs, such as the one shown in Tbl. 1, to complete the diagram and derive a behavioral specification of the system.

## 3 Preliminaries

Let $S$ be a set. The powerset of $S$ is denoted by $\mathcal{P}(S) = \{S' \mid S' \subseteq S\}$. We use $|S|$ for the number of elements in $S$. Two sets $U$ and $V$ are *disjoint* if $U \cap V = \emptyset$. Some set $S$ with relation $\leqslant$ is a partial order, denoted by $(S, \leqslant)$, iff $\leqslant$ is reflexive, i.e. $a \leqslant a$ for all $a \in S$, antisymmetric, i.e. $a \leqslant b$ and $b \leqslant a$ imply $a = b$ for all $a, b \in S$, and transitive, i.e. $a \leqslant b$ and $b \leqslant c$ imply $a \leqslant c$ for all $a, b, c \in S$. Given a relation $R \subseteq S \times S$ for some set $S$, we denote its transitive closure by $R^+$, and the transitive and reflexive closure by $R^*$.

A *bag* $m$ over $S$ is a function $m : S \to I\!N$, where $I\!N = \{0, 1, \ldots\}$ denotes the set of natural numbers. We denote e.g. the bag $m$ with an element $a$ occurring once, $b$ occurring three times and $c$ occurring twice by $m = [a, b^3, c^2]$. The set of all bags over $S$ is denoted by $I\!N^S$. Sets can be seen as a special kind of bag where all elements occur only once; we interpret sets in this way whenever we use them in operations on bags. We use $+$ and $-$ for the sum and difference of two bags, and $=$, $<$, $>$, $\leqslant$, $\geqslant$ for the comparison of two bags, which are defined in a standard way.

A *sequence* over $S$ of length $n \in I\!N$ is a function $\sigma : \{1, \ldots, n\} \to S$. If $n > 0$ and $\sigma(i) = a_i$ for $i \in \{1, \ldots, n\}$, we write $\sigma = \langle a_1, \ldots, a_n \rangle$. The length of a sequence is denoted by $|\sigma|$. The sequence of length 0 is called the *empty sequence*, and is denoted by $\epsilon$. The set of all finite sequences over $S$ is denoted by $S^*$. We write $a \in \sigma$ if a $1 \leqslant i \leqslant |\sigma|$ exists such that $\sigma(i) = a$. *Concatenation* of two sequences $\nu, \gamma \in S^*$, denoted by $\sigma = \nu; \gamma$, is a sequence defined by $\sigma : \{1, \ldots, |\nu| + |\gamma|\} \to S$, such that $\sigma(i) = \nu(i)$ for $1 \leqslant i \leqslant |\nu|$, and $\sigma(i) = \gamma(i - |\nu|)$ for $|\nu| + 1 \leqslant i \leqslant |\nu| + |\gamma|$. A sequence $\sigma$ can be projected over some set $U$, denoted by $\sigma_{|U}$, and is inductively defined by $\epsilon_{|U} = \epsilon$, $(\langle a \rangle; \sigma)_{|U} = \langle a \rangle; \sigma_{|U}$ if $a \in U$, and $(\langle a \rangle; \sigma)_{|U} = \sigma_{|U}$ otherwise.

**Petri Nets** A *Petri net* [16] is a tuple $N = \langle P, T, F \rangle$ where (1) $P$ and $T$ are two disjoint sets of *places* and *transitions* respectively; and (2) $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*. The elements from the set $P \cup T$ are called the *nodes* of $N$. Elements of $F$ are called *arcs*. Places are depicted as circles, transitions as squares. For each element $(n_1, n_2) \in F$, an arc is drawn from $n_1$ to $n_2$.

Let $N = \langle P, T, F \rangle$ be a Petri net. Given a node $n \in (P \cup T)$, we define its *preset* ${}^\bullet_N n = \{n' \mid (n', n) \in F\}$, and its *postset* $n^\bullet_N = \{n' \mid (n, n') \in F\}$. We lift the notation of preset and postset to sets. Given a set $U \subseteq (P \cup T)$, ${}^\bullet_N U = \bigcup_{n \in U} {}^\bullet_N n$ and $U^\bullet_N = \bigcup_{n \in U} n^\bullet_N$. If the context is clear, we omit the $N$ in the subscript.

A *marking* of $N$ is a bag $m \in \mathbb{N}^P$, where $m(p)$ denotes the number of *tokens* in place $p \in P$. If $m(p) > 0$, place $p$ is called *marked* in marking $m$. A Petri net $N$ with corresponding marking $m$ is written as $(N, m)$ and is called a *marked Petri net*. Given a marked Petri net $(N, m)$, transition $t$ is enabled, denoted by $(N, m)[t\rangle$, if ${}^\bullet t \leqslant m$. If transition $t$ is enabled in $(N, m)$, it can fire, resulting in a new marking $m'$, denoted by $(N, m)[t\rangle(N, m')$, such that $m' + {}^\bullet t = m + t^\bullet$. We lift the firing of transitions to the firing of sequences in a standard way, i.e., a sequence $\sigma \in T^*$ of length $n$ is enabled in $(N, m)$ if markings $m_0, \ldots, m_n$ exist, such that $m = m_0$ and $(N, m_{i-1})[\sigma(i)\rangle(N, m_i)$ for all $1 \leqslant i \leqslant n$. A marking $m'$ is reachable from some marking $m$ in $N$, denoted by $(N, m)[*\rangle(N, m')$, if a firing sequence $\sigma \in T^*$ exists such that $(N, m)[\sigma\rangle(N, m')$. A marking $m'$ is a *home marking* of $(N, m)$, if for all markings $m''$ with $(N, m)[*\rangle(N, m'')$, we have $(N, m'')[*\rangle(N, m')$.

A special class of Petri nets are the workflow nets [1]. A workflow net is a tuple $\langle P, T, F, i, f \rangle$ with $\langle P, T, F \rangle$ a Petri net, (2) $i \in P$ is the only place with no incoming transitions, (3) $f \in P$ is the only place with no outgoing transitions, i.e., ${}^\bullet i = f^\bullet = \emptyset$, and (4) all transitions have at least one incoming and one outgoing arc, i.e., ${}^\bullet t \neq \emptyset \neq t^\bullet$ for all $t \in T$.

**Open Petri Nets** Within a network of asynchronously communicating systems, messages are passed between the elements within the network. The approach we follow is based on Open Petri nets [5]. Communication in an open Petri net (OPN) is represented by special places, called the *interface places*. An interface place is either an *input place*, receiving messages from the outside, or an *output place* that sends messages to the outside of the OPN. An input place is a place that has only outgoing arcs, and an output place has no incoming arcs.

**Definition 1.** *An* Open Petri net *is defined as an 7-tuple* $\langle P, I, O, T, F, i, \Omega \rangle$ *where (1)* $\langle P \cup I \cup O, T, F \rangle$ *is a Petri net; (2)* $P$ *is a set of* internal places*; (3)* $I$ *is a set of* input places*, and* ${}^\bullet I = \emptyset$*; (4)* $O$ *is a set of* output places*, and* $O^\bullet = \emptyset$*; (5)* $P$*,* $I$ *and* $O$ *are pairwise disjoint; (6)* $i \in \mathbb{N}^P$ *is the* initial marking*, and (7)* $\Omega \subseteq \mathbb{N}^P$ *is the set of* final markings*. We call the set* $I \cup O$ *the* interface places *of the OPN. An OPN is called* closed *if* $I = O = \emptyset$*.*

An important behavioral property for OPNs is termination: an OPN should always have the possibility to terminate properly. We identify two termination properties: weak termination and soundness.

**Definition 2.** *Let $\langle P, I, O, T, F, i, f \rangle$ be an OPN. It is* weakly terminating*, if for every reachable marking of the marked Petri net $(\langle P \cup I \cup O, T, F \rangle, i)$ a marking $f \in \Omega$ can be reached.*

*It is* sound*, if for every reachable marking of the marked Petri net $(\langle P, T, F \rangle, i)$ a marking $f \in \Omega$ can be reached.*

Communication between OPNs is done via the interface places. Two OPNs can only communicate if the input places of the one are the output places of the other, and vice versa.

**Definition 3.** *Two OPNs $A$ and $B$ are* composable*, denoted by $A \oplus B$, if and only if $(I_A \cap O_B) \cup (O_B \cap I_A) = (P_A \cup T_A \cup I_A \cup O_A) \cap (P_B \cup T_B \cup I_B \cup O_B)$.*

*If $A$ and $B$ are composable, they can be composed into a new OPN, denoted by $A \oplus B$, with $A \oplus B = \langle P, I, O, T, F, i, \Omega \rangle$ where $P = P_A \cup P_B \cup G$; $I = (I_A \cup I_B) \backslash G$; $O = (O_A \cup O_B) \backslash G$; $T = T_A \cup T_B$; $F = F_A \cup F_B$; $i = i_A + i_B$; and $f = \Omega_A \cup \Omega_B$ with $G = (I_A \cap O_B) \cup (O_B \cap I_A)$.*

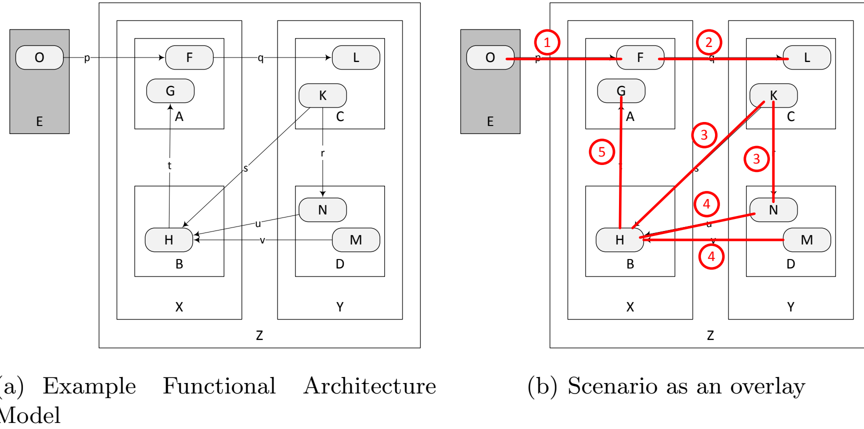
**Event Logs and Behavioral Profiles** Although event logs are defined as a tuple consisting of a set of case identifiers, events, and an attribute mapping [2], it is in this paper sufficient to consider an event log, denoted by $\mathcal{L}$, as a set of sequences over some alphabet $T$, i.e., $\mathcal{L} \subseteq T^*$. Given an event log $\mathcal{L}$, we define the *successor relation* [20] by $a <_{\mathcal{L}} b$ if a sequence $\sigma \in \mathcal{L}$ and $1 \leqslant i \leqslant |\sigma|$ exist, such that $\sigma(i) = a$ and $\sigma(i+1) = b$. Using the successor relation, we define the *behavioral profile* $(\rightarrow_c, \|_c, +_c)_{\mathcal{L}}$ as three relations: (1) the causality relation $\rightarrow_c$ is defined by $a \rightarrow_c b$ iff $a <_{\mathcal{L}} b$ and $b \nless_{\mathcal{L}} a$, (2) the concurrency relation $\|_c$, which is defined by $a \|_c b$ iff both $a <_{\mathcal{L}} b$ and $b <_{\mathcal{L}} a$, and (3) the exclusive relation $+_c$ is defined by $a +_c b$ iff both $a \nless_{\mathcal{L}} b$ and $b \nless_{\mathcal{L}} a$ [20]. If the context is clear, we omit the subscript.

Given a marked Petri net $(N, m)$ with $N = \langle P, T, F \rangle$, an event log $\mathcal{L} \subseteq T^*$ is called *complete* with respect to $(N, m)$ iff traces $\sigma_1, \sigma_2 \in T^*$ exist such that $(N, m)[\sigma_1; \langle a, b \rangle; \sigma_2 \rangle (N,)$ implies $a <_{\mathcal{L}} b$ for all $a, b \in T$.

## 4 Functional Architectures

To model the overview of a system, the modules it consists of, and the features these modules offer, we propose the use of the *functional architecture model* (FAM). The functional architecture of a system is "an architectural model which represents at a high level the software products major functions from a usage perspective, and specifies the interactions of functions, internally between each other and externally with other products" [6]. It offers modules containing features. Features of different modules interact via so-called *information flows*.

An example is shown in Fig. 2(a). The FAM contains 1 context module, $E$, 7 modules, $A$, $B$, $C$, $D$, $X$, $Y$ and $Z$. Modules have features, depicted by the rounded rectangles. For example, module $C$ contains two features, $K$ and $L$. Between features of different modules, information flows exist, e.g., the information flow $(F, q, L)$ between modules $A$ and $C$.




(a) Example Functional Architecture Model

(b) Scenario as an overlay

**Fig. 2.** Example Functional Architecture Model and corresponding scenario as overlay

**Definition 4.** *A* Functional Architecture Model *(FAM) is defined as a 6-tuple* $\langle \mathcal{M}, \mathcal{C}, \mathcal{F}, h, m, \rightarrow \rangle$ *where*

- $\mathcal{M}$ *is a finite set of* modules*;*
- $\mathcal{C}$ *is a finite set of* context modules*;*
- $\mathcal{F}$ *is a finite set of* features*;*
- $h : \mathcal{M} \rightarrow \mathcal{M}$ *is the* hierarchy function*, such that the transitive closure* $h^*$ *is irreflexive;*
- $m : \mathcal{F} \rightarrow \mathcal{M} \cup \mathcal{C}$ *is a* feature map *that maps each feature to a module, possibly in the context, and this module does not have any children, i.e.* $h^{-1}(m(F)) = \emptyset$ *for all* $F \in \mathcal{F}$*;*
- $\rightarrow \subseteq \mathcal{F} \times \Lambda \times \mathcal{F}$ *is the* information flow*, with* $\Lambda$ *the label universe, such that for* $(A, l, B) \in \rightarrow$ *we have* $m(A) \neq m(B)$*. The labels for the information flows are unique per feature, i.e.,* $(A, l, B)$ *and* $(A, l, C)$ *imply* $B = C$ *for all labels* $l \in \Lambda$ *and* $(A, l, B), (A, l, C) \in \rightarrow$*.*

Although the information flows define the possible interactions between modules, it remains a static overview of the system. Therefore, one can use scenarios on top of the functional architecture, e.g. by creating an overlay, highlighting the information flows that are executed and the order in which they should occur. Formally, we represent a scenario as a partial order.

**Definition 5.** *Let* $F = \langle \mathcal{M}, \mathcal{C}, \mathcal{F}, h, m, \rightarrow \rangle$ *be a FAM. A* scenario *of* $F$ *is a pair* $(S, <)$ *with* $S \subseteq \rightarrow$*, such that* $(S, \leqslant)$ *with* $\leqslant = <^*$ *is a partial order.*

An example is shown in Fig. 2(b). The scenario implied by the overlay can be represented by a partial order induced by $(O, p, F) < (F, q, L)$, $(F, q, L) < (K, s, H)$, $(F, q, L) < (K, r, N)$, $(K, r, N) < (N, u, H)$, $(K, s, H) < (H, t, G)$, $(N, u, H) < (H, t, G)$, $(K, r, N) < (M, v, H)$, and $(M, v, H) < (H, t, G)$.

However, such scenarios are typically not specified. Another important drawback of such scenarios is their analyzability. Although each scenario can be checked, the consistency between the different scenarios remains a difficult task. Therefore, in the remainder of this paper, we search for a method to derive the behavioral specification as a network of asynchronously communicating systems, given the system execution data produced by the actual system in the form of event logs.

## 5 Discovery of a Functional Architecture

In this section, we study the possibilities process mining [2] offers to generate Petri nets for each of the different modules a system consists of. Event logs describe the order in which features of a system have been executed. Such event logs are system wide. Instead of each module having its own event log, only global sequences exist, i.e., sequences concatenate the executed features over all modules. As FAM only allows features to be contained in a single module, we assume that each feature belongs to exactly one module. Also, FAM prescribes communication to be one-directional, i.e., given two communicating features $A$ and $B$, we assume that either $A$ sends a message to $B$, or vice versa, that $B$ sends a message to $A$, but not both.

The behavioral specification of a system is three-fold: (1) communication between modules via their features, (2) the internal behavior within each feature, and (3) the order in which features are called within a module. In this section, we explore all three types of behavioral specification to come to a composed system of asynchronously communicating systems.

In the remainder, let $\mathcal{L}$ be an event log over a set of features $T$, and let $\mathfrak{R} : T \to M$, with $M$ the set of modules, be a function that maps each feature onto the module that contains that feature.

### 5.1 Communication between Features

Communication between modules within a system is asynchronous of nature: messages are sent between features in order to complete their functionality. Within an event log, we need to consider the order in which events or features occur. For example, given some trace $\sigma$, if the resource is different for two subsequent events, i.e., $\mathfrak{R}(\sigma(i)) \neq \mathfrak{R}(\sigma(i+1))$, then this might indicate that the former sends a message to the latter. This is expressed by the communication successor.

**Definition 6 (Communication successor).** *Let $\mathcal{L} \subseteq T^*$ be an event log. We define the* communication successor *relation $\lll_{\mathcal{L}} \subseteq T \times T$ by $A \lll_{\mathcal{L}} B$ iff $\mathfrak{R}(A) \neq \mathfrak{R}(B)$, $\sigma(i) = A$, and $\sigma(i+1) = B$ for some $\sigma \in \mathcal{L}$ and $1 \leqslant i < |\sigma|$.*

Although at first sight the communication successors seem to work, we need to remember the concurrent nature of asynchronous communication. Consider for example the communication between modules $M$ and $N$ as depicted in Fig. 3, and

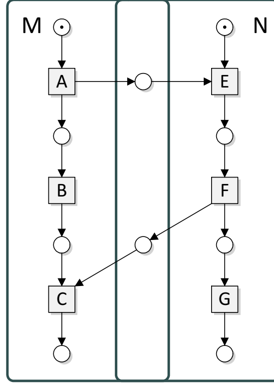



**Fig. 3.** Modules $M$ and $N$

| Case | Trace |
|---|---|
| 1 | A, E, F, G, B, C |
| 2 | A, E, F, B, G, C |
| 3 | A, E, B, F, G, C |
| 4 | A, E, B, F, C, G |
| 5 | A, E, F, B, C, G |
| 6 | A, B, E, F, G, C |
| 7 | A, B, E, F, C, G |

**Table 2.** Corresponding event log

| | **E** | **F** | **G** |
|---|---|---|---|
| **A** | $\rightarrow$ | $+$ | $+$ |
| **B** | $\parallel$ | $\parallel$ | $\parallel$ |
| **C** | $+$ | $\leftarrow$ | $\parallel$ |

**Table 3.** Communication behavioral profile

the corresponding allowed sequences in Tbl. 4. We have $A \lessdot E$, which is indeed the communication as modeled in the composition $M \oplus N$. However, we also find $G \lessdot B$, indicating a possible communication between $G$ and $B$. Listing all communication successors, we get $A \lessdot E$, $G \lessdot B$, $F \lessdot B$, $B \lessdot G$, $G \lessdot C$, $E \lessdot B$, $B \lessdot F$, $F \lessdot C$, $C \lessdot G$, and $B \lessdot E$. Observe that because of the asynchronous nature of the communication, features $B$ and $E$ are concurrently enabled in Fig. 3. Assuming the event log to be complete, this should become visible in the communication successor relation, as for the normal successor relation on event logs.

**Definition 7 (Communication behavioral profile).** *Let $\mathcal{L} \subseteq T^*$ be an event log, and $\lessdot_{\mathcal{L}} \subseteq T \times T$ the corresponding communication successor relation.*

*The* communication behavioral profile *is the 3-tuple $(\rightarrow_c, \|_c, +_c)^{Com}_{\mathcal{L}}$ defined by:*

- *$A \rightarrow_c B$ iff $A \lessdot_{\mathcal{L}} B$ and $B \not\lessdot_{\mathcal{L}} A$;*
- *$A \|_c B$ iff both $A \lessdot_{\mathcal{L}} B$ and $B \lessdot_{\mathcal{L}} A$; and*
- *$A +_c B$ iff both $A \not\lessdot_{\mathcal{L}} B$ and $A \not\lessdot_{\mathcal{L}} B$.*

Calculating the behavioral profile of the communicating transitions using the communication successor relation, results in the communication behavioral profile as shown in Tbl. 3. It shows that $B$ and $E$ are concurrently enabled. Following the behavioral profile, we see that the causal relation of the behavioral profile correctly identifies the feature communication.

Using the communication behavioral profile, we can construct the information flows from an event log as follows. If $A \rightarrow B$ in the communication behavioral

| | Debtor | | | | | | | Payment | | | | | | | Creditor | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | F | G | H | I | J | K | O | M | N | Q | S |
| A | | | | | | | | $\leftarrow$ | + | + | + | + | + | + | + | + |
| B | | | | | | | | + | $\rightarrow$ | + | + | + | + | + | + | + |
| C | | | | | | | | + | + | $\rightarrow$ | + | + | + | + | + | + |
| D | | | | | | | | + | + | + | $\leftarrow$ | + | + | $\Vert$ | + | + |
| E | | | | | | | | + | + | + | + | $\leftarrow$ | + | + | + | + |
| F | | | | | | | | + | + | + | + | + | $\rightarrow$ | + | + | + |
| G | | | | | | | | + | $\Vert$ | + | + | + | $\Vert$ | $\Vert$ | + | $\Vert$ |
| H | $\rightarrow$ | + | + | + | + | + | + | | | | | | | | + | + |
| I | + | $\leftarrow$ | + | + | + | + | $\Vert$ | | | | | | | | + | + |
| J | + | + | $\leftarrow$ | + | + | + | + | | | | | | | | $\rightarrow$ | + |
| K | + | + | + | $\rightarrow$ | + | + | + | | | | | | | | + | $\leftarrow$ |
| O | + | + | + | + | $\rightarrow$ | + | + | | | | | | | | + | $\leftarrow$ |
| M | + | + | + | + | + | $\leftarrow$ | $\Vert$ | | | | | | | | + | $\rightarrow$ |
| N | + | + | + | $\Vert$ | + | + | $\Vert$ | | | | | | | | + | $\Vert$ |
| Q | + | + | + | + | + | + | + | + | + | $\leftarrow$ | + | + | + | + | | |
| S | + | + | + | + | + | + | $\Vert$ | + | + | + | $\rightarrow$ | $\rightarrow$ | $\leftarrow$ | $\Vert$ | | |

**Table 4.** Communication behavioral profile for the running example

profile of the event log, then an information flow $(A, x, B)$ exists, with $x$ a fresh label. This results in the following translation:

**Definition 8 (Generated FAM).** *Let $\mathcal{L}$ be an event log, and $(\rightarrow_c, \Vert_c, +_c)^{Com}_{\mathcal{L}}$ be its communication behavioral profile. Its corresponding functional architecture model $\langle \mathcal{M}, \mathcal{C}, \mathcal{F}, h, m, \rightarrow \rangle$ is defined by:*

- $\mathcal{M} = \mathfrak{R}(\mathcal{L})$;
- $\mathcal{C} = \varnothing$;
- $\mathcal{F} = T$;
- $h = \varnothing$;
- $m = \mathfrak{R}$; *and*
- $\rightarrow = \{(A, x, B) \mid A \rightarrow_c B,\ \text{and } x \in \Lambda \text{ a fresh label}\}$.

After constructing the communication behavioral profile for the running example, shown in Tbl. 4, we can complete the functional architecture model. Based on the given system execution data, we see for example that feature $H$ communicates with feature $A$, and feature $S$ sends messages to features $K$ and $O$, and receives messages from feature $M$. The complete functional architecture of the running example is shown in Fig. 4.

### 5.2 Internal Behavior of Features

As can be seen in the running example, features can send and receive multiple messages. For example, feature $S$ sometimes sends a message to feature $K$ and sometimes to feature $O$. Therefore, the next step in discovering the functional

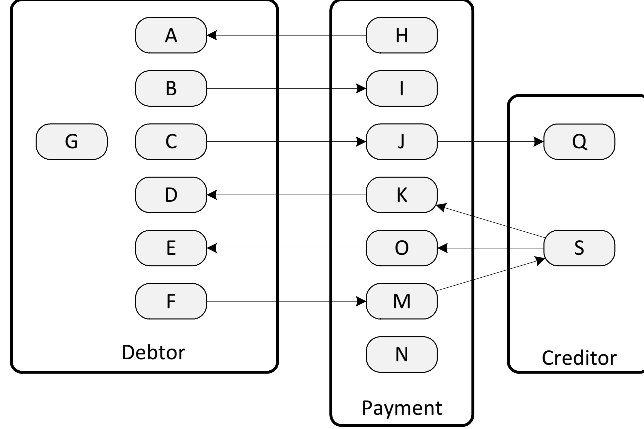



**Fig. 4.** Functional architecture model of the running example

architecture is to reconstruct the internal behavior of each of the features. For this, we create for each of the features an event log, containing the features that it communicates with. We call this the *feature log.*

**Definition 9 (Feature log).** *Let $\mathcal{L} \subseteq T^*$ be an event log, and let $F \in T$ be some feature. Let $(\rightarrow_c, \|_c, +_c)_{\mathcal{L}}^{Com}$ be the corresponding communication behavioral profile. The* feature log *$\mathcal{L}_F$ is defined by $\mathcal{L}_F = \{\sigma_{|C(F)} \mid \sigma \in \mathcal{L}, F \in \sigma\}$ where $C(F) = \{A \mid A \rightarrow_c F \vee F \rightarrow_c A\}$.*

Consider for example feature $S$ in the running example. This feature communicates with features $K$, $O$ and $M$, i.e., $C(S) = \{K, O, M\}$. Its feature log is the projection of the log on these features, i.e., $\mathcal{L}_S = \{\langle K\rangle, \langle O, M\rangle\}$.

On these feature logs, we apply the inductive miner [13], that always returns a sound workflow net. Next, we transform the discovered workflow net into an open Petri net, to visualize the messages sent and received by the feature. This results in a *feature net* for each of the features present in the event log.

**Definition 10 (Feature Net).** *Let $\mathcal{L} \subseteq T^*$ be an event log, and let $F \in T$ be some feature. Let $(\rightarrow_c, \|_c, +_c)_{\mathcal{L}}^{Com}$ be the corresponding communication behavioral profile. The* Feature net *$\mathcal{N}_F$ is the OPN $\langle P, I, O, T, F, i, \Omega\rangle$ defined by*

- $P = \bar{P}$, $T = \bar{T}$, $i = [\,\bar{i}\,]$, $\Omega = \{\ [\,\bar{f}\,]\ \}$;
- $I = \{p_{A-F} \mid A \rightarrow_c F\}$;
- $O = \{p_{F-A} \mid F \rightarrow_c A\}$;
- $F = \bar{F} \cup \{(t, p_{F-A}) \mid t \in T, \lambda(t) = A, F \rightarrow_c A\}$
  $\cup \{(p_{A-F}, t) \mid t \in T, \lambda(t) = A, A \rightarrow_c F)\}$.

*where $\langle \bar{P}, \bar{T}, \bar{F}, \bar{i}, \bar{f}\rangle$ is the discovered workflow net.*

In our running example, each of the 16 features are transformed into a feature net. Most of the features are simple, like for feature $H$ and $A$, consisting of

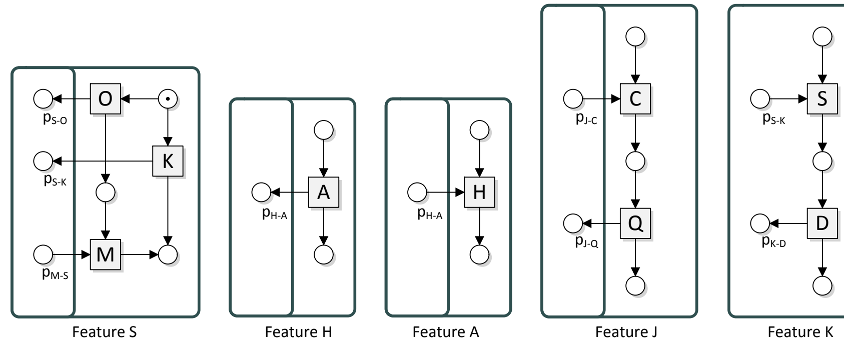



**Fig. 5.** Some of the feature nets for the running example

a single transition sending a message to $A$, and receiving a message from $H$, respectively. A more complex feature net is the net for feature $S$, which internally decides whether it sends a message to $K$ or to $O$. Figure 5 depicts some of the feature nets generated using the inductive miner [13].

### 5.3 Feature Interaction within Modules

Now that each feature has its internal behavior defined by means of a feature net, the next step is to determine the order in which features are executed within each of the modules. As for the features, we first create event logs for each of the modules, by filtering each trace on the features it contains. This results in a *module log* for each of the modules.

**Definition 11 (Module Log).** *Let $\mathcal{L} \subseteq T^*$ be an event log. Let $M \in Rng(\mathfrak{R})$ be a module. Let $(\rightarrow_c, \|_c, +_c)$ be the corresponding communication behavioral profile. The* Module log $\mathcal{L}_M$ *is defined by $\mathcal{L}_M = \{\sigma_{|\{F|\mathfrak{R}(F)=M\}} \mid \sigma \in \mathcal{L}\}$.*

Within the running example, we obtain three module logs, one for each of the modules. For example, module *Debtor*, has module log $\mathcal{L}_{\text{Debtor}} = \{\langle A, B, G\rangle$, $\langle A, C, D, G\rangle$, $\langle A, C, E, F, G\rangle\}$, and for *Creditor* we have $\mathcal{L}_{\text{Creditor}} = \{\langle Q, S\rangle$,

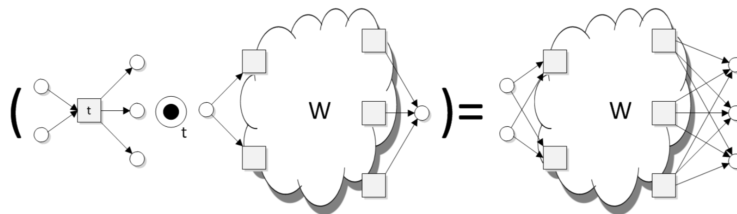



**Fig. 6.** Refinement of a transition by a workflow net

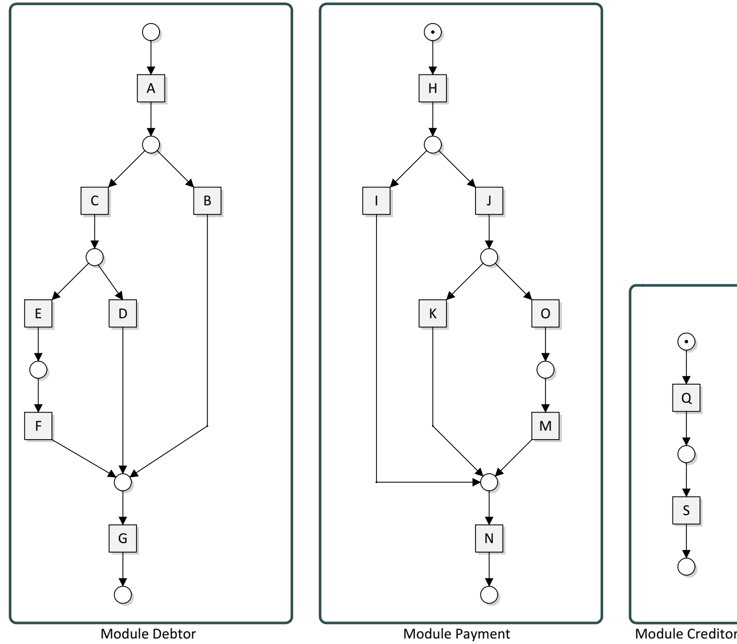



**Fig. 7.** Module nets for the running example

$\langle Q, S, S\rangle\}$. Again applying the inductive miner results in the three workflow nets as depicted in Fig. 7. Notice that, although feature $S$ occurs twice in one of the sequences, the algorithm only adds a single feature $S$ in the resulting workflow model.

### 5.4 Composition of Feature Nets and Module Nets

Last step in the process is to combine the feature nets generated for each of the features with the generated module nets. This results in an open Petri net for each of the modules, defining the interaction between the different modules.

In the module net, each feature is represented by a single transition. Next step is to refine each feature by its feature net. For this, we first define the refinement of a transition by a workflow model on open Petri nets, as shown in Fig. 6. This refinement connect each input place of the refined transition with each of the transitions in the postset of the initial place of the workflow, and similarly each output place of the refined transition with each of the transitions in the preset of the final place of the refining workflow. It is straight-forward to prove that if (1) the initial net is sound, (2) each input place of the refined transition is 1-bounded, i.e., it can contain at most one token, and (3) workflow net $W$ is sound, then the refinement yields a sound result.

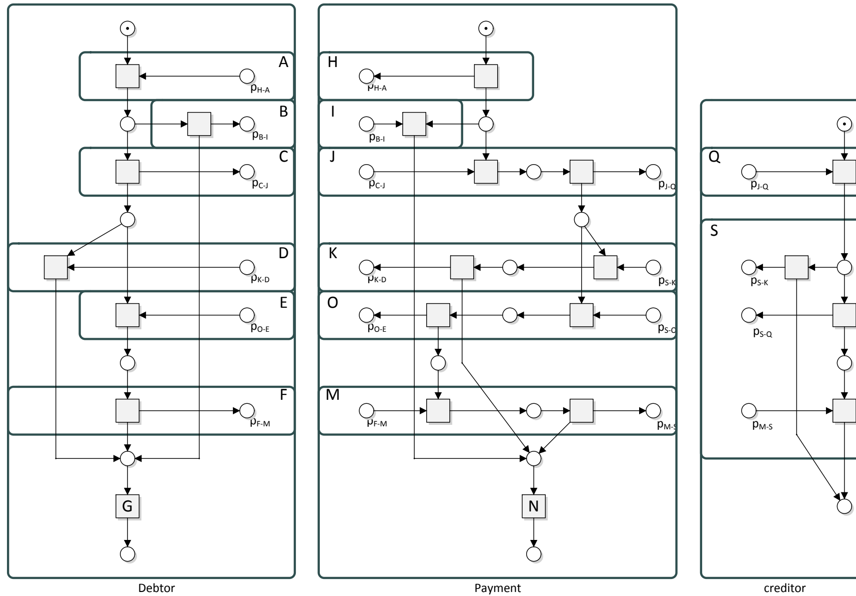



**Fig. 8.** Composed nets generated for the running example

The result of refining each feature by its feature net is shown in Fig. 8. As features $G$ and $N$ have no feature net defining communication, these transitions are not refined.

To verify whether the resulting open Petri nets are a true representation of the system, one can compose the nets into a single Petri net, and execute each of the sequences of the event log of Tbl. 1 on the resulting model, which in this example is possible. Further analyzing the resulting model shows that its only deadlocks are desirable markings: either all modules reach their final place, without any pending tokens, or the *Creditor* module remains untouched, while the *Debtor* and *Payment* module reach their final place.

## 6 Conclusions

Within this paper, we discussed a method to automatically generate a functional architecture model from an event log together with a mapping of each feature to the module that offers that functionality. We showed how the information flows can be derived from the communication behavioral profile. This profile not only identifies the information flow for the static structure of the functional architecture, but additionally offers sufficient information to construct the internal behavior for each of the features, and between the features within a module. Lastly, we showed how to compose feature and module nets into an open Petri net.

Discovering the interaction between different modules is not new. Tchniques like service mining [3], apply process mining on event logs to discover a process model of how the services are orchestrated. In the approach presented in this paper, we focus on the discovery of the behavior of each of the modules, rather than a complete orchestration.

In [15], the authors discover the internal behavior of services based on the interaction between two services, guaranteeing deadlock freedom of the discovered service. In the setting of this paper, the exact interaction between modules is unknown, and needs to be discovered first.

The core idea of this paper is twofold: firstly to derive the information flows for a Functional Architecture Model, and secondly to derive the internal behavior for each of the modules within the architecture. Within software architecture, this is called Software Architecture Reconstruction [12]. Although some techniques take the dynamic aspects of the software operation into account, most techniques only focus on the static aspects of software architecture models, using solely the available source code [8]. For example, system execution data is used to enrich architectures with performance data [11] or to visualize traces on how the software is used [19]. In this paper, we propose a method to not only visualize software usage, but to discover module communication and to generate the internal behavior of modules within a software architecture.

Although the approach presented in this paper shows an application of the behavioral profile to discover feature interaction, additional research is required. First, the current approach requires the event log to be complete, i.e., if the log grows, the successor relation should not change. Further, for the generation of the internal feature behavior, we assume that if the sending feature is present in the event log, it enables all possible events, which is possibly a too strict assumption that deserves further investigation.

The approach in this paper is very flexible, as we derive individual models for the features and modules. For this, we apply standard process discovery algorithms returning sound workflow models. However, their composition in general does not result in a sound system of asynchronously communicating systems. Further research is required to study the conditions under which this can be guaranteed. For this, we want to identify conditions which on the one hand result in correct models, and on the other hand have a positive effect on model quality as described by [7].

Not only does this approach provide useful insights for the software architect, we expect the approach applicable to business process management as well, as for the discovery of separate business processes, the Business Process Modelling and Notation offers the swimlane notion. Therefore, we plan to implement the approach in the Process Mining toolkit ProM [18] to experiment and apply the approach on real-life examples.

**Acknowledgements** The authors would like to thank the anonymous reviewers for their valuable feedback, and Sjaak Brinkkemper, Fabiano Dalpiaz, Garm Lucassen, Leo Pruijt and Erik Jagroep for the fruitfull discussions and valuable input on architecture and software products.

## References


1. W.M.P. van der Aalst. Verification of workflow nets. In *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407 – 426. Springer, Berlin, 1997.
2. W.M.P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, Berlin, 2011.
3. W.M.P. van der Aalst. Challenges in service mining: Record, check, discover. In *Web Engineering*, volume 7977 of *Lecture Notes in Computer Science*, pages 1–4. Springer, Berlin, 2013.
4. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Series in Software Engineering. Addison Wesley, Reading, MA, USA, 2012.
5. D. Bera, K. M. van Hee, and J.M.E.M. van der Werf. Designing weakly terminating ros systems. In *Applications and Theory of Petri Nets, (33th International Conference, Petri Nets 2012)*, volume 7347 of *Lecture Notes in Computer Science*, pages 328 – 347. Springer, Berlin, 2012.
6. S. Brinkkemper and S. Pachidi. Functional architecture modeling for the software product industry. In *ECSA 2010*, volume 6285 of *Lecture Notes in Computer Science*, pages 198 – 213. Springer, Berlin, 2010.
7. J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. On the role of fitness, precision, generalization and simplicity in process discovery. In *On the Move to Meaningful Internet Systems: OTM 2012*, volume 7565 of *Lecture Notes in Computer Science*, pages 305–322. Springer, Berlin, 2012.
8. S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *Software Engineering, IEEE Transactions on*, 35(4):573–591, July 2009.
9. S.A. Fricker. Software product management. In *Software for People*, Management for Professionals, pages 53–81. Springer, 2012.
10. K.M. van Hee, N. Sidorova, and J.M.E.M. van der Werf. When can we trust a third party? In *Transactions on Petri Nets and Other Models of Concurrency VIII*, volume 8100 of *Lecture Notes in Computer Science*, pages 106–122. Springer, Berlin, 2013.
11. T. Israr, M. Woodside, and G. Franks. Interaction tree algorithms to extract effective architecture and layered performance models from traces. *Journal of Systems and Software*, 80(4):474 – 492, 2007. Software Performance 5th International Workshop on Software and Performance.
12. R.L. Krikhaar. *Software Architecture Reconstruction*. PhD thesis, VU Amsterdam, 1999.
13. S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst. Discovering block-structured process models from event logs - a constructive approach. In *Application and Theory of Petri Nets and Concurrency*, volume 7927 of *Lecture Notes in Computer Science*, pages 311–329. Springer, Berlin, 2013.
14. G. Lucassen, J.M.E.M. van der Werf, and S. Brinkkemper. Alignment of software product management and software architecture with discussion models. In *IWSPM 2014*, pages 21–30. IEEE, Aug 2014.
15. R. Müller, C. Stahl, W.M.P. van der Aalst, and M Westergaard. Service discovery from observed behavior while guaranteeing deadlock freedom in collaborations. In *Service-Oriented Computing*, volume 8274 of *Lecture Notes in Computer Science*, pages 358–373. Springer, Berlin, 2013.
16. W. Reisig. *Petri Nets: An Introduction*, volume 4 of *Monographs in Theoretical Computer Science: An EATCS Series*. Springer, Berlin, 1985.

17. R.N. Taylor, N. Medvidovic, and E.M. Dashofy. *Software Architecture: Foundations, Theory, and Practice.* John Wiley & Sons, 2010.
18. H.M.W. Verbeek, J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. XES, XESame, and ProM 6. In *Information System Evolution*, volume 72 of *Lecture Notes in Business Information Processing*, pages 60–75. Springer, Berlin, 2011.
19. R.J. Walker, G.C. Murphy, J. Steinbok, and M.P. Robillard. Efficient mapping of software system traces to architectural views. In *Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '00, pages 12–. IBM Press, 2000.
20. M. Weidlich and J.M.E.M. van der Werf. On profiles and footprints – relational semantics for petri nets. In *Applications and Theory of Petri Nets (ICATPN 2012)*, volume 7347 of *Lecture Notes in Computer Science*, pages 148 – 167. Springer, Berlin, 2012.
21. J.M.E.M. van der Werf and H.M.W. Verbeek. Online compliance monitoring of service landscapes. In *BPM 2014 International Workshops*, volume 202 of *Lecture Notes in Business Information Processing*, pages 89–95. Springer, Berlin, 2015.