Computer Science and Information Systems 18(3):657–686 https://doi.org/10.2298/CSIS200131013B

Reverse Engineering Models of Software Interfaces [⋆]

Debjyoti Bera[1], Mathijs Schuts[2], Jozef Hooman[3], and Ivan Kurtev[4]

[1] ESI (TNO), The Netherlands
debjyoti.bera@tno.nl
[2] Philips, Best, The Netherlands
mathijs.schuts@philips.com
[3] ESI (TNO) and Radboud University, The Netherlands
jozef.hooman@tno.nl
[4] Altran and Eindhoven University of Technology (TU/e), The Netherlands
i.kurtev@tue.nl

Abstract. Cyber-physical systems consist of many hardware and software components. Over the lifetime of these systems their components are often replaced or updated. To avoid integration problems, formal specifications of component interface behavior are crucial. Such a formal specification captures not only the set of provided operations but also the order of using them and the constraints on their timing behavior. Usually the order of operations are expressed in terms of a state machine. For new components such a formal specification can be derived from requirements. However, for legacy components such interface descriptions are usually not available. So they have to be reverse engineered from existing event logs and source code. This costs a lot of time and does not scale very well. To improve the efficiency of this process, we present a passive learning technique for interface models inspired by process mining techniques. The approach is based on representing causal relations between events present in an event log and their timing information as a timed-causal graph. The graph is further processed and eventually transformed into a state machine and a set of timing constraints. Compared to other approaches in literature which focus on the general problem of inferring state-based behavior, we exploit patterns of client-server interactions in event logs.

Keywords: passive learning, process mining, interfaces, model-driven engineering.

1. Introduction

The high-tech industry creates complex cyber-physical systems. These systems consist of many hardware and software components. These components are either developed inhouse or made by third party suppliers. Components interact with each other over software interfaces. Good interface descriptions are crucial for component-based development of cyber-physical systems. Usually software interfaces are only described in terms of their signature, i.e., the set of operations. Sometimes also the allowed sequence of operations is specified, for instance in terms of a state machine or a few example scenarios. The timing behavior of an interface are rarely described. For instance, the expected frequency of notifications and the allowed time between the call of an operation and the corresponding

[⋆] This is a revised and extended version of the conference paper [39].

response. Violations of assumptions about timing behavior, however, are an important source of errors over the complete life cycle of these systems.

To overcome the drawbacks of current interface modeling tools, we have developed an Eclipse-based DSL (Domain Specific Language) called ComMA (Component Modeling and Analysis). ComMA [26] is currently used at business unit Image Guided Therapy (IGT) of Philips for specifying interfaces of software components. A ComMA interface specification describes the signature and behavior of a server component. The signature of a server captures the operations it offers to clients and the notifications that it can send to clients. The behavior of a server is specified as a state machine describing the allowed sequence of interactions available to a client. Next to such a state machine, timing constraints on interactions, and data constraints on parameters exchanged during interactions can also be specified. Based on a ComMA specification, a large number of artifacts are generated automatically, for example visualization of the modeled state machine, interface design documentation, simulator, stubs and run-time monitors. The monitor is very useful to check interface compliance after software updates or hardware upgrades. For instance, this is useful when an updated hardware component is obtained from a supplier, or during integration of software developed by different teams. Often during software development, teams are working on different sub-systems with shared interfaces. To facilitate teams to work independently and reduce the chances of costly integration issues later, stubs are particularly useful.

Given the benefits of the ComMA approach, all new system interfaces of Philips IGT are modeled and checked using ComMA. However, there are many existing interfaces and they could benefit from a ComMA specification. A manual transformation based on inspecting event logs and software artifacts (source code, documentation etc.) would require a large reverse engineering effort, is hard to scale up and extremely error prone. On the other hand, the idea of automatically inferring state-based behavior from event logs (also known as passive learning) is not new. This topic has been extensively studied by the two communities of finite state machine inference [11] and process mining [1, 29].

Within both these communities there are popular tools that provide a large variety of techniques to infer automata or Petri net models from event logs. However it is difficult for a non-expert to use these tools directly on event logs to infer interface protocols. For instance, the relation between inferred models and the choice of techniques (and their many configuration parameters and heuristics) is not always clear and may require a deep understanding of these techniques. There is also the additional effort required to translate the output models into a more meaningful domain specific representation. Furthermore, most passive learning approaches focus on ordering of events (actions) but neglect data and time aspects. Moreover, the special case of inferring interfaces of component-based systems can benefit from exploiting patterns of client-server interactions present in event logs. To address these challenges, we present a method to infer component interface models (state machines) and their timing constraints from event logs.

Our approach is based on process mining techniques that exploit the ordering relation between events of an event log (such as α-algorithm [2] and inductive miner [30]). Such techniques usually start by computing causal dependencies between events and represent them as a causal graph (vertices correspond to events). We extend a causal graph to capture both data and time information present in an event log. From a causal graph, we derive a state machine graph and timing constraints over events. Finally, we reduce the state machine graph by merging equivalent states and transform it into a ComMA state machine. At various stages, we exploit patterns of client-server interactions in event logs such as recurring events and compound events to achieve better generalizations in the resulting model.

In contrast, the approach in our previous paper [39], first transforms an event log into a type of Moore automata [24] and then into a ComMA state machine and a set of time constraints. The idea in that paper is to identify *event groups* that start with a client-initiated event (command or signal) followed by zero or more server-initiated events (notifications). Each event group is mapped to a ComMA triggered transition. Such grouping of events often leads to a large number of states since variations in the number and type of notifications will result in different event groups. The approach is also not able to deal with event logs starting with a notification, nor is it correctly able to discover compound events.

Concerning other model learning techniques, we have experimented earlier with active learning which stimulates the system under learning actively and infers an hypothesis based on the responses of the system [41]. Active learning [6] requires the implementation of an adapter to connect the System Under Learning (SUL) with the learner. This adapter has to deal with behavior of the SUL that does not match the assumptions of the learning techniques, such as a SUL which is not input enabled or a SUL which sends no output or multiple outputs after a stimulus. This technique also requires frequent resets of the SUL which may be time consuming. Furthermore, non-determinism of the SUL is a problem for active learning.

A disadvantage of passive learning is that only the behavior that is represented in the used traces will be in the resulting state machine. Hence, compared to the active learning approach, the model might be less complete. An interesting approach presented in [49] exploits the complementary nature of the results produced by passive learning and active learning to improve the final outcome. In our case, however, a passive learning approach is acceptable since the learned model is intended as a starting point for subsequent manual modeling and analysis.

Structure of this paper

The paper is organized as follows. Section 2 provides a brief overview of interface specifications using ComMA. In Section 3 we present our automated workflow to reverse engineer interface models from event logs. Section 4 presents our learning method to infer state and timing behavior. Section 5 describes a few extensions to the learning algorithm. In Section 6, we apply our learning method to two real-life cases at Philips and evaluate the generated models by comparing them to the original models. In Section 7 related work is discussed and we compare our approach to a popular state merging approach for inferring finite state machines. Section 8 concludes the paper.

2. Model-Based Definition of Interfaces in ComMA

In this section, we introduce ComMA [26] as far as needed to understand the remainder of this paper. The ComMA framework consists of the following three main languages: signature, interface and traces. We illustrate the languages using a rather simple example of a vending machine interface called IVendingMachine.

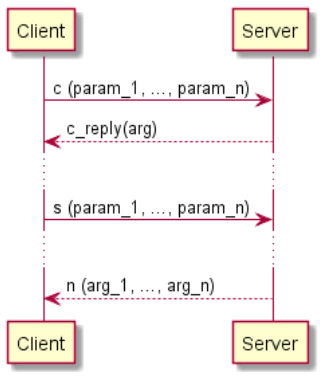


Fig. 1. UML Sequence Diagram of possible event types between a client and server

```
signature IVendingMachine {
    types
    enum Result { OK, NOK }

    commands
    void loadProduct
    Result switchOn
    Result insertCoin
    Result orderCola

    signals
    switchOff

    notifications
    inventoryInfo(int items)
}
```

Listing 1.1. Example of a signature

ComMA Signature. A ComMA signature specification lists a set of events offered by a server to its clients. The events of a signature are distinguished into three types:

- *Commands* are synchronous events from client to server. The client is blocked until it receives a *reply* from the server.
- *Signals* are asynchronous events from client to server.
- *Notifications* are asynchronous events from server to client.

All event types may have data associated with them. Their type definitions are also specified in a signature. To describe data aspects, ComMA provides a set of primitive data types (such as integer, string, boolean, real etc.) and allows the definition of more complex types such as enumerated types, vectors and records.

We refer to data associated with a command or a signal as *input parameters* and data associated with a reply or a notification as *output arguments*.

In the Fig. 1, we give one example of each type of event in the execution context of a client and server. Observe command c and its corresponding reply c_reply. The command c has n input parameters associated with it denoted by *param_1*, $\ldots$, *param_n*. The reply of command c denoted by *c_reply* has one output argument *arg* associated with it.

```
interface IVendingMachine{
  variables
    int items, coins

  init
    items := 0
    coins := 0

  in all states {
        transition do: inventoryInfo(items)
  }

  initial state Off {
        transition trigger: loadProduct
        do: items := items + 1
        reply
        next state: Off

        transition trigger: switchOn
        guard: items > 0
        do: reply(OK)
        next state: On

        transition trigger: switchOn
        guard: items <= 0
        do: reply(NOK)
        next state: Off
  }

  state On {
        transition trigger: insertCoin
        do: coins := coins + 1
        reply(OK)
        next state: On
        OR
        do: reply(NOK)
        next state: On

        transition trigger: orderCola
        guard: coins > 0 AND items > 0
        do: coins := coins - 1
        items := items - 1
        reply(OK)
        next state: On

        transition trigger: orderCola
        guard: coins <= 0 OR items <= 0
        do: reply(NOK)
        next state: On

        transition trigger: switchOff
        next state: Off
  }
}
```

Listing 1.2. Example of a ComMA state machine

Note that signals and notifications do not have a corresponding reply. In listing 1.1 we present the signature of IVendingMachine.

ComMA Interface. The behavior of an interface in terms of allowed sequence of events is expressed in terms of state machines. A state machine has one or more states (with ex-

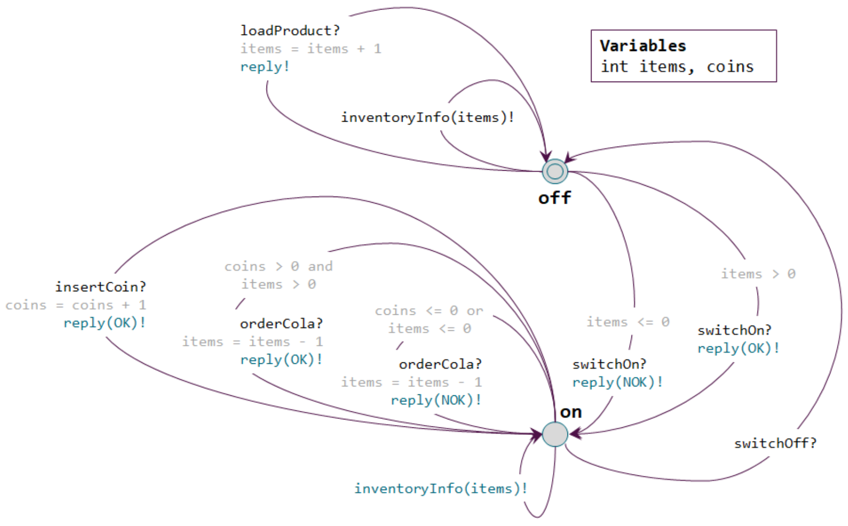


Fig. 2. Server side state machine corresponding Listing 1.2

actly one initial state) and a set of declared and initialized variables. Each state must have one or more transitions. We distinguish between triggered and non-triggered transitions. Both kinds may have an associated guard over the set of defined variables. Triggered transitions (denoted by *transition trigger*) represent an invocation by a client, i.e. either a command or a signal. Non-triggered transitions (denoted by *transition*) represent an event emitted by the server, i.e. notification or reply to a command. The body of a transition consists of one or more *clauses* separated by an OR construct. A *clause* is a sequences of *actions* corresponding variables assignments (using standard mathematical expressions) or events (notifications or replies to commands). Non-determinism between choice of possible transitions in a state is supported by simply defining multiple transitions. Within a transition body we express non-determinism using the OR construct.

We present the interface state machine of our IVendingMachine example in Listing 1.2. A brief explanation is provided below:

– Two variables $items$ and $coins$ are defined and initialized.
– The initial state is Off.
– Notification *inventoryInfo* with parameter $items$ is possible in all states.
– In state Off there is a choice between accepting commands *loadProduct* and *switchOn*. Observe that there are two instances of transition *switchOn* with different guards.
– In state On there is a non-deterministic choice between accepting commands *insertCoin* and *orderCola* and a signal *switchOff*. Observe that *insertCoin* has two possible replies, expressed by the OR construct and different reply values.

In the Fig. 2, we present the communicating state machine (server side) corresponding Listing 1.2 by borrowing the syntax of the popular modeling tool UPPAAL [28] for communicating automata. We extend the notation on arcs between states to represent sequence of communicating events with expressions on variables.

```
timing constraints

TimingRule1
command orderProduct and reply(OK) -> [ 2.4 ms ..  3.8 ms ] between events

TimingRule2
notification inventoryInfo and notification inventoryInfo
    -> [ 400.0 ms .. 550.0 ms ] between events
```

Listing 1.3. Example of a few timing constraints

```
Timestamp: 0.000081 Notification: inventoryInfo Parameter: integer : 0
Timestamp: 2.002300 Notification: inventoryInfo Parameter: integer : 0

Timestamp: 3.030400 Command: switchOn
Timestamp: 3.567788 Reply Parameter: Result::NOK

Timestamp: 4.005600 Command: loadProduct
Timestamp: 4.206180 Reply

Timestamp: 5.640320 Notification: inventoryInfo Parameter: integer : 1
Timestamp: 6.940301 Notification: inventoryInfo Parameter: integer : 1

Timestamp: 7.046780 Command: switchOn
Timestamp: 7.666180 Reply Parameter: Result::OK

Timestamp: 13.100550 Command: orderCola
Timestamp: 13.215671 Reply Parameter: Result::NOK

Timestamp: 17.705500 Notification: inventoryInfo Parameter: integer : 1
Timestamp: 18.905500 Notification: inventoryInfo Parameter: integer : 1

Timestamp: 19.055012 Command: insertCoin
Timestamp: 20.000020 Reply Parameter: Result::NOK

Timestamp: 23.100550 Command: orderCola
Timestamp: 23.215671 Reply Parameter: Result::NOK

Timestamp: 23.908800 Notification: inventoryInfo Parameter: integer : 1
Timestamp: 24.608703 Notification: inventoryInfo Parameter: integer : 1
Timestamp: 25.888101 Notification: inventoryInfo Parameter: integer : 1

Timestamp: 26.000300 Command: insertCoin
Timestamp: 26.006180 Reply Parameter: Result::OK

Timestamp: 27.100550 Command: orderCola
Timestamp: 27.215671 Reply Parameter: Result::OK

Timestamp: 28.030440 Notification: inventoryInfo Parameter: integer : 0
Timestamp: 29.960241 Notification: inventoryInfo Parameter: integer : 0

Timestamp: 30.000300 Signal: switchOff

Timestamp: 36.000330 Command: switchOn
Timestamp: 36.006180 Reply Parameter: Result::NOK
```

Listing 1.4. Fragment of a ComMA trace

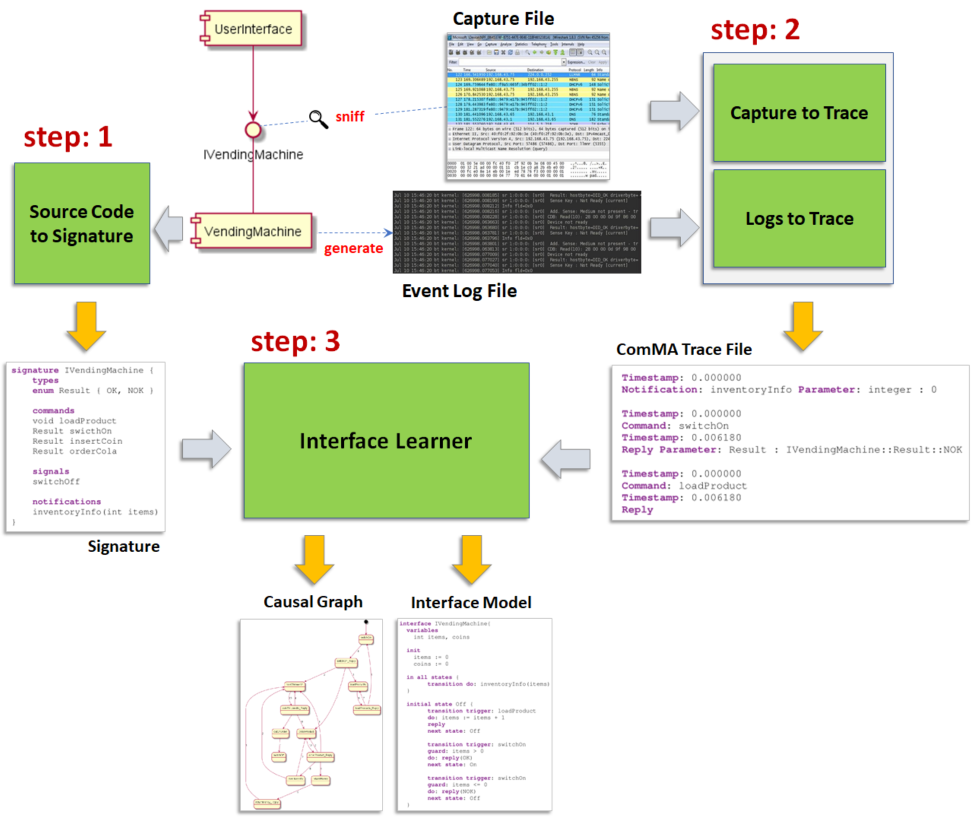
Timing Constraints Next to a state machine specification, a ComMA interface also allows the specification of timing behavior as a set of timing constraints. In ComMA there are four types of timing constraints [25], but we will only consider three of them.

Interval rules specify the time interval between events. *Conditional interval rules* are a further restriction on them. *Rules for periodic events* specify repetitive occurrence of an event within a specified time period and jitter.

Listing 1.3 shows two examples of timing constraints: *TimingRule1* describes the allowed time between an occurrence of command *orderProduct* and its *reply*. The lower bound (LB) is 2.4 ms and the upper bound (UB) is 3.8 ms. *TimingRule2* gives another example of how time intervals between periodic notifications are similarly specified.

ComMA Trace. The trace language in ComMA is used to represent observed interactions between a client and a server in a language independent manner. The idea is to be able to write custom converters from domain specific traces (e.g. sniffing network traffic or from a generated log file) to the ComMA trace language. An example ComMA trace conforming to the IVendingMachine interface is shown in Listing 1.4.

3. Reverse Engineering Interfaces of Legacy Systems



```
signature IVendingMachine {
    types
    enum Result { OK, NOK }

    commands
    void loadProduct
    Result swicthOn
    Result insertCoin
    Result orderCola

    signals
    switchOff

    notifications
    inventoryInfo(int items)
}
```

```
Timestamp: 0.000000
Notification: inventoryInfo Parameter: integer : 0

Timestamp: 0.000000
Command: switchOn
Timestamp: 0.006180
Reply Parameter: Result : IVendingMachine::Result::NOK

Timestamp: 0.000000
Command: loadProduct
Timestamp: 0.006180
Reply
```

```
interface IVendingMachine{
  variables
    int items, coins

  init
    items := 0
    coins := 0

  in all states {
        transition do: inventoryInfo(items)
  }

  initial state Off {
        transition trigger: loadProduct
        do: items := items + 1
        reply
        next state: Off

        transition trigger: switchOn
        guard: items > 0
        do: reply(OK)
        next state: On

        transition trigger: switchOn
        guard: items <= 0
        do: reply(NOK)
        next state: Off
  }
```

Fig. 3. Typical Usage of our Learning Framework

Often the behavior of legacy components is poorly documented and understood. This makes it hard to create an interface model having the right level of abstraction. In the previous section, we have discussed the two ingredients of a ComMA interface model, namely signature and interface. In Fig. 3, we present the different steps to derive a ComMA interface model.

Generating a signature model from available source code is rather easy (step 1). On the other hand, inferring interface behavior in terms of a state machine requires run-time information such as event logs. Often event logs are abundantly available but the information in them may not be very useful due to unclear semantics and incompleteness. In such cases sniffing network channels during system execution and extracting information from them turns out to be a very useful technique to obtain consistent and complete execution data. In all cases, we must create custom transformations to the ComMA trace format (step 2). A prerequisite for performing step 2 is the ability to map the custom messages and their data to the available ComMA event types (as used in signatures) and ComMA data types. ComMA provides the commonly used primitive types such as integer, real, string, enumerations, records, vectors and map types. Our experience shows that these data types are generally sufficient to support non-trivial cases. A limitation of ComMA is the inability to use references to interface instances or services as data types (a feature that can be found in some protocols for distributed computing). The actual conversion from the captured communication or logs to ComMA traces usually requires developing a custom translator. Depending on the complexity of the protocol and the data format, this task may lead to significant efforts. In our practice we have created custom translators that deal with proprietary company specific protocols, and also faced mixed formats that integrate textual, binary and sometimes encrypted data. Clearly, availability of documentation about the protocol is a key enabling factor. Usage of standard protocols and existing tools can greatly reduce the effort in step 2.

The *Interface Learner* is an implementation of the reverse engineering method presented in Sec. 4. Once we have the set of ComMA traces and the signature model, the interface learner generates a timed-causal graph and a ComMA state machine containing time constraints (step 3).

Causal graphs are widely used by many commercial process mining tools [31] [5] as a simple and intuitive means to visualize which activities in a trace can follow one another directly [6]. Most of these tools nicely capture time and frequency based information such as execution times and activity counts.

4. Inferring State Behavior: Interface Learner

We present a step-wise method to transform an event log into a ComMA state machine. First we represent the information in an event log as a *causal graph* (dependency graph) of events extended with time. A *causal graph* is then transformed into a *state machine graph* where edges are events and states are outputs of events (data), i.e. similar to Moore automata [24]. Next all *equivalent* states (i.e. states having same set of possible events) of a state machine graph are discovered and merged. Finally the resulting state machine graph is transformed into ComMA state machine syntax using a simple algorithm.

[5] https://www.celonis.com/, https://processgold.com/en/, https://www. my-invenio.com/

[6] See https://www.gartner.com/doc/3870291/market-guide-process-mining

First we introduce a few notations in Sec. 4.1. In Sec. 4.2 we describe our learning method in steps: logs to causal graphs, causal graphs to state machines, merging equivalent states and finally generating a ComMA interface model and a set of timing constraints.

4.1. Notations

General definitions A finite *sequence* σ over some set S of length $n \in \mathbb{N}$ is a function $\sigma : \{1, \ldots, n\} \rightarrow S$. The set of all finite sequences over S is denoted by S^*. We denote a sequence of length n by $\sigma = \langle s_1, \ldots, s_n \rangle$, where $s_1, \ldots, s_n \in S$ and $\sigma(i) = s_i$ for $1 \leq i \leq n$. A sequence of length 0 is called an empty sequence denoted by ϵ.

A *graph* is a pair $G = (V, E)$, where V is the set of vertices and relation $E \subseteq V \times V$ denotes edges. In a directed graph, edges have directions represented by a head and tail. In a directed graph, a sequence $\sigma \in V^*$ of length $n > 0$ is called a *directed path*, if $(\sigma(i), \sigma(i+1)) \in E$ for all $1 \leq i < n$.

We assume a set of *interface actions IA*, consisting of commands, replies, signals and notification, with a function $type$ which yields the type of each action, that is, *COMMAND*, *SIGNAL*, *REPLY*, or *NOTIFICATION*. For a command c we denote its reply by c_reply. Henceforth we use $a, a_1, a_2, \ldots$ to denote interface actions, $c, c_1, c_2, \ldots$ to denote commands, $s, s_1, s_2, \ldots$ to denote signals, and $n, n_1, n_2, \ldots$ to denote notifications.

An *event* is a tuple (a, str), where a is an interface action and str is an *output string* which is a string representation of the value of one or more *output arguments* associated with a reply or notification. For commands and signals the output string is empty (see Sec. 2). For now we abstract away from *input arguments* of commands and signals but revisit it later in this section.

Given an event e, we denote its interface action by $action(e)$ and its output string by $output(e)$. The occurrence of an event e at time $t \in \mathbb{R}^+$ is called a *timed event*, denoted as the pair (e, t). We define the projection functions $event(e, t) = e$ and $time(e, t) = t$. Let Σ be the set of all timed events. A *trace* σ is a possibly empty sequence over Σ, $\sigma \in \Sigma^*$. We denote the empty trace by ϵ.
As an example,

$$\sigma = \langle ((c, \text{-}), 0.0); ((c\_reply, \text{OK}), 0.215); ((s, -), 3.51); ((n, 5), 4.11) \rangle$$

is a trace containing first command c at time 0.0 followed by its reply with value OK at 0.215, third signal s at 3.51 and fourth notification n with output 5 at 4.11.

A *log* L is a finite non-empty set of traces $L \subseteq \Sigma^*$. For a given log L, we define $events(L)$ as the set of all *events* occurring in traces of L and $actions(L)$ as the set of all interface actions occurring in $events(L)$.

Restriction [R1] In this section we require for every trace that every occurrence of a command is immediately followed by a reply of this command, that is, there are no intermediate notifications. In Section 5, we will discuss how this restriction can be relaxed.

4.2. The Learning Algorithm

The learning algorithm takes a log and a signature as input and produces an interface model as output. First we convert a log to a causal graph which is later transformed to a state machine.

Logs to Causal Graphs Each trace of a given log L captures a possible order of the occurrences of events in time. A *causal graph* is a directed graph which describes when two events follow each other in a trace. For a log L we define its causal graph $G(L)$ as the graph (V, E) where

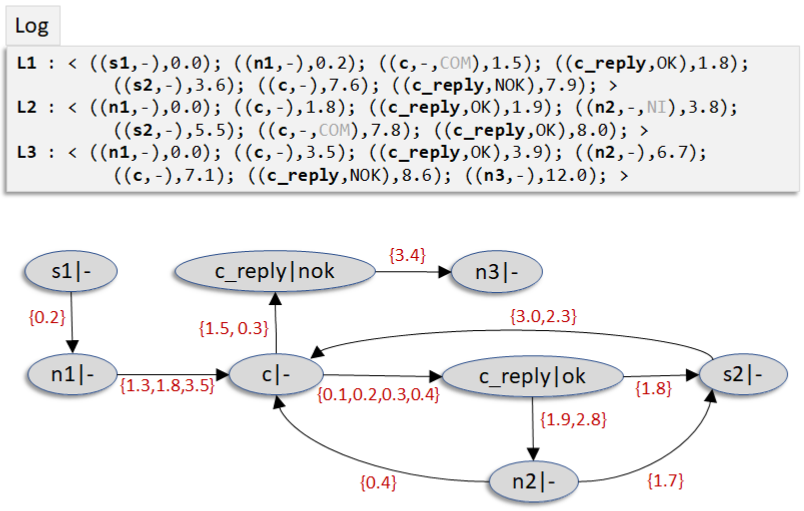
- $V = events(L)$
- $(e, e') \in E$ if and only if there exists a trace $\langle te_1, te_2, \ldots te_n\rangle \in L$ and an $i \in \{1, \ldots, n-1\}$ such that $e = event(te_i)$ and $e' = event(te_{i+1})$.

We denote the set of *initial vertices* as $initial(V) = \{\{event(\sigma(0))\} \mid \sigma \in L\}$.

We associate a set of time durations to each pair of causally related events by the function $\delta : E \to \mathcal{P}(\mathbb{R}^+)$ which is defined as follows:

$$\begin{aligned}\delta(e, e') = \{t \mid & \text{ there exists a trace } \langle te_1, te_2, \ldots te_n\rangle \in L \text{ and an } i \in \{1, \ldots, n-1\} \\ & \text{such that } e = event(te_i),\ e' = event(te_{i+1}) \text{ and} \\ & t = time(te_{i+1}) - time(te_i)\}\end{aligned}$$

We refer to the pair $(G(L), \delta)$ as a *timed causal graph*. An example of a log containing three traces and its corresponding timed causal graph is shown in Fig. 4. The edges are annotated with the set of time durations, as defined by function δ.

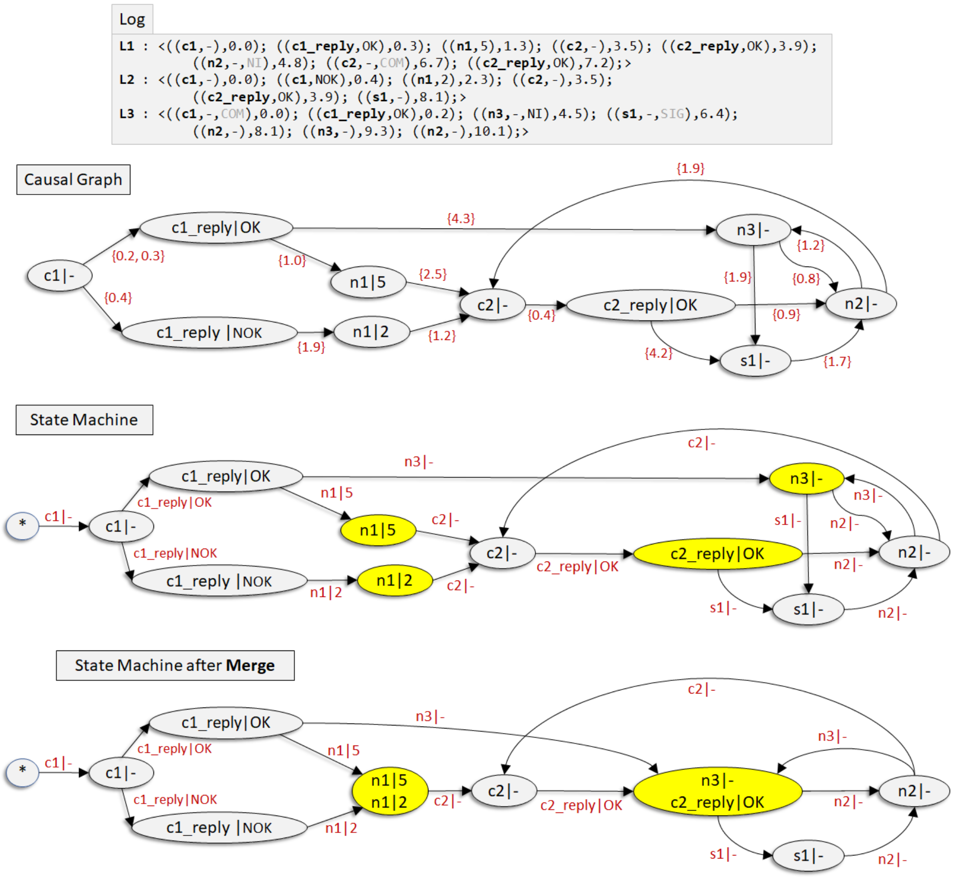


```
Log
L1 : < ((s1,-),0.0); ((n1,-),0.2); ((c,-,COM),1.5); ((c_reply,OK),1.8);
       ((s2,-),3.6); ((c,-),7.6); ((c_reply,NOK),7.9); >
L2 : < ((n1,-),0.0); ((c,-),1.8); ((c_reply,OK),1.9); ((n2,-,NI),3.8);
       ((s2,-),5.5); ((c,-,COM),7.8); ((c_reply,OK),8.0); >
L3 : < ((n1,-),0.0); ((c,-),3.5); ((c_reply,OK),3.9); ((n2,-),6.7);
       ((c,-),7.1); ((c_reply,NOK),8.6); ((n3,-),12.0); >
```

Fig. 4. Three Traces and their Timed Causal Graph

Causal Graphs to State Machines Next we transform a causal graph into a state machine graph, where each state (vertex) represents a set of events and each transition (edge) represents an event. An event of an incoming transition to a state belongs to the set of events associated with that state.

A *state machine* is a tuple (S, A, T, s_0), where S is the set of states, A is the set of actions, $T \subseteq S \times A \times S$ is the set of transitions, and s_0 is the initial state.



Log

```
L1 : <((c1,-),0.0); ((c1_reply,OK),0.3); ((n1,5),1.3); ((c2,-),3.5); ((c2_reply,OK),3.9);
        ((n2,-,NI),4.8); ((c2,-,COM),6.7); ((c2_reply,OK),7.2);>
L2 : <((c1,-),0.0); ((c1,NOK),0.4); ((n1,2),2.3); ((c2,-),3.5);
        ((c2_reply,OK),3.9); ((s1,-),8.1);>
L3 : <((c1,-,COM),0.0); ((c1_reply,OK),0.2); ((n3,-,NI),4.5); ((s1,-,SIG),6.4);
        ((n2,-),8.1); ((n3,-),9.3); ((n2,-),10.1);>
```

Fig. 5. Causal Graph to State Machine

We do not include time durations in a state machine because a timed causal graph is sufficient to derive timing constraints. This is discussed later in the section.

Given a log L and its causal graph $G(L) = (V, E)$, we define a *state machine* $(S, A, T, init)$ where

- $init \notin V$
- $S = \{\{e\} | e \in events(L)\} \cup \{init\}$, i.e. states different from $init$ are singleton sets, each corresponding a distinct event from log L
- $A = events(L)$;
- $T = \{(s_1, e, s_2) \mid e \in s_2$ and $((s_1, s_2) \in E$ or $s_2 \in initial(V) \wedge s_1 = \{init\})\}$
  Note that transitions with source $init$ are added to all *initial vertices* of V. The event of a transition is obtained from the target state of the relation in E.

An example of a causal graph and its corresponding state machine is shown in the Fig. 5. Note that we denote the state $init$ by the node containing the symbol $*$.

Algorithm 1: Generate ComMA StateMachine

```
input  : state machine $SM = (S, A, T, s_0)$, name
output: ComMA state machine
print machine name {
for $s \in S$ do
    if $s = s_0$ then
        print initial state StateName$(s)$ {
    else
        print state StateName$(s)$ {
    end
    for $(s, e, s_1) \in T$ do
        if $e$ has-type NOTIFICATION then
            print transition do: EventName$(e)$
            print next state: StateName$(s_2)$
        end
        if $e$ has-type SIGNAL then
            print transition trigger: EventName$(e)$
            print next state: StateName$(s_2)$
        end
        if $e$ has-typeCOMMAND then
            print transition trigger: EventName$(e)$
            print do:
            for $path \in$ getAllPathsToReply$(SM, s)$ do
                for $(s_1, e, s_2) \in path$ do
                    if $e$ has-type REPLY then
                        print EventName$(e)$
                        print next state: StateName$(s_2)$
                        if not last for-iteration over paths then
                            print OR
                        end
                    else
                        print EventName$(e)$
                    end
                end
            end
        end
    end
    print}
end
print }
```

Merging Equivalent States Once we have a state machine, the goal is to *reduce* it by iteratively discovering all pairs of equivalent states and merging them.

Given a state machine (S, A, T, s_0), two states $s_1, s_2 \in S$ are said to be *equivalent* if and only if $\{a \mid \exists s : (s_1, a, s) \in T\} = \{a \mid \exists s : (s_2, a, s) \in T\}$, i.e., the same set of events are possible.

In the Fig. 5, consider the two pairs of states $\{(n1, 5)\}$ and $\{(n1, 2)\}$, $\{(n3, -)\}$ and $\{(c2\_reply, \text{OK})\}$. It is easy to check that both pairs are *equivalent*. Each state of the first

```
interface ISample{

  initial state init {
        transition trigger: c1
        do: reply(OK)
        next state: executed_c1_OK
        OR
        do: reply(NOK)
        next state: executed_c1_NOK
     }

    state executed_c1_OK {
        transition do: n1(5)
        next state: executed_n1_5_n1_2

        transition do: n3
        next state: executed_n3_c2_OK
    }

    state executed_c1_NOK {
        transition do: n1(2)
        next state: executed_n1_5_n_2
    }

    state executed_n1_5_n1_2 {
        transition trigger: c2
        do: reply(OK)
        next state: executed_n3_c2_OK
    }

    state  executed_n3_c2_OK {
        transition do: n2
        next state: executed_n2

        transition trigger: s1
        next state: executed_s1

    }

    state executed_s1 {
        transition do: n2
        next state: executed_n2
    }

    state executed_n2 {
        transition trigger: c2
        do: reply(OK)
        next state: executed_n3_c2_OK

        transition do: n3
        next state: executed_n3_c2_OK
    }
}
```

Listing 1.5. Generated ComMA State Machine

pair allows a single event $(c2, -)$ (with destination state: $\{(c2, -)\}$), while each state of the second pair allows two events $(s1, -)$ (with destination state: $\{(s1, -)\}$) and $(n2, -)$ (with destination state: $\{(n2, -)\}$).

Given a state machine (S, A, T, s_0) and two equivalent states $s_1, s_2 \in S$, where $s_2 \neq init$, we define a *merge* operation *Merge*$((S, A, T, s_0), s_1, s_2) = (S', A, T', s'_0)$ where

- $S' = S \setminus \{s_1, s_2\} \cup \{s\}$ where s is the union of s_1 and s_2
- T' is obtained from T by replacing all occurrences of s_1 and s_2 by s
- $s'_0 = s$, if $s_1 = init$, $s'_0 = s_0$, otherwise.

It is easy to check that the *merge* operation does not disturb the set of possible paths in the state machine (i.e. action sequences). Also note that the order of applying the *merge* operation does not have an effect on the resulting state machine.

Generating ComMA Interface Model A state machine $SM = (S, A, T, s_0)$ can be transformed into a ComMA interface model, where we assume the following methods:

- *getAllPathsToReply*(SM, s) returns the set of all paths of SM starting at state s and ending with an event of type *REPLY* and no other event in this path is of type *REPLY*. Due to restriction $R1$, a command event is immediately followed by a reply event.
- *StateName*(s) which yields a meaningful string representation (label) of state s, for instance, as a disjunction of event output strings over all incoming edges (see state labels in Fig. 5).
- *EventName*(e) which yields a string representation of event e.

Given a state machine, we present an algorithm (see Alg. 1) to generate its corresponding ComMA interface model. For the last state machine (i.e. after merging) in Fig. 5, the algorithm produces the output shown in Listing.1.5.

Remarks. Often data sets associated with a notification or reply belong to large domains such as integers, real etc. They may also have a complex type such as records and vectors. In such cases the number of resulting states may become very large since the output strings are not equal (for e.g. see notification $n1$ with output 5 and 2 in the Fig. 5). To remedy this we may abstract away from such data sets. This should ideally be indicated by the user as part of configuration parameters of the learner. If we abstract away from all data in notifications and reply (i.e. all output strings are empty), then the size of the resulting state machine is bounded by the number of unique events in an event log. This is easy to check since the number of vertices in a causal graph and its corresponding state machine (excluding *init*) are bounded by that number.

Recall that we did not consider input arguments of commands and signals. It is straightforward to capture them in a similar manner as we did for output data of replies and notifications. To achieve this we only need to extend the tuple representing an *event* to be the tuple $(a, ostr, istr)$, where a is an interface action (as before), $ostr$ is an output string (as before) and $istr$ is an input string which is a string representation of one or more argument values associated with a command or signal. Similar to the previous case, abstraction techniques are needed to avoid an explosion of states.

It is easy to check that the resulting state machine conforms to the input event log since (a) the causal graphs represent all possible ordering of events present in an event log, and (b) the corresponding merged state machine is not disturbing the order of events present in the causal graph. To validate the implementation of our learning algorithm, the monitoring feature in ComMA is used to check for *conformance* between the generated interface model and the set of input event logs [26].

Generating timing constraints. Consider a log L and its timed causal graph $((V, E), \delta)$. From a timed causal graph we could generate all possible timing constraints between pairs of events but not all of them will be useful from a functional requirements point of view, for e.g. between two status reporting notifications or between an error and a status notification etc. Rather timing constraints over response time of an operation to execute a movement on a mechanical device (observable as a specific command and its eventual reply) or between notifications reporting positions of the device are more interesting to check for compliance with safety regulations.

Recall from Sec. 3 that one of the inputs to the interface learner is the signature file containing the syntactic definitions of each unique event occurring in event logs. So as part of the configuration parameters of the interface learner, the user has the possibility to indicate which set of events in the signature are relevant and what *types* of timing constraints over them would be useful to generate.

As pre-requisite for generating timing constraints from a timed causal graph, we assume a few generic methods are present to compute minimum and maximum time durations between events (user indicates which events are useful) present in a log. The notations for these methods are described below.

- Given a command event $c \in V$, we denote the minimum time duration to observe any of its reply events as $c_{min} = \min(\{\min(\delta(c, r)) \mid (c, r) \in E\})$ and maximum time duration to observe any of its reply events as $c_{max} = \max(\{\max(\delta(c, r)) \mid (c, r) \in E\})$.
- Given two events $e1, e2 \in V$ such that $e2$ is reachable from every path starting at $e1$ and there are no cycles in between, let Γ be the set of all paths starting at $e1$ and ending at $e2$. We denote the cumulative minimum and maximum time durations along all paths of Γ by the interval $[e1e2_{min}, e1e2_{max}]$. For the special case of cyclic paths where $e = e1 = e2$, we denote its period by $e_{period} = (e_{max} - e_{min})/2$ and jitter by $e_{jitter} = \max(e_{max} - e_{period}, e_{period} - e_{min})$.

In Listing. 1.6, we provide templates to generate three types of timing constraints. A brief description of these is given below (for detailed semantics, see [25]):

- *TimeForReply* states that if command c is observed then its reply is observed within the specified interval.
- *TimeBetweenEvents* states that if two events $e1$ and $e2$ are observed without observing $e1$ in between then the interval between them is [$e1e2_{min}$ ms ... $e1e2_{max}$ ms].
- *TimeBetweenPeriodicEvents* states that if $e1$ is observed then $e2$ will occur periodically with period $e2_{period}$ and jitter $e2_{jitter}$ until $e3$ observed.

As mentioned earlier, the user indicates the events $c, e1, e2, e3$ as part of configuration parameters of the learner. The Listing 1.7 shows four examples of timing constraints generated from the timed causal graph presented in Fig. 4.

5. Extensions

The learning algorithm presented in the previous section is very general in that it does not exploit patterns of interaction between a client and its server. We present two extensions to the learning algorithm to deal with some commonly occurring patterns.

```
timing constraints

TimeForReply
command $action(c)$
    -[ $c_{min}$ ms ... $c_{max}$ ms ] -> reply

TimeBetweenEvents
$type(action(e1))$ $action(e1)$ and $type(action(e2))$ $action(e2)$
    -> [ $e1e2_{min}$ ms ... $e1e2_{max}$ ms ] between events

TimeBetweenPeriodicEvents
$action(e1)$ then $action(e2)$ with period $e_{period}$ ms jitter $e_{jitter}$ ms until $action(e3)$
```

Listing 1.6. Templates to generate timing constraints

```
timing constraints

TimingConstraint1
command $c$ -[ 0.1 ms ... 1.5 ms ] -> reply

TimingConstraint2
signal $s1$ and command $c$ -> [ 1.5 ms ... 3.7 ms ] between events

%TimingConstraint3
%notification $n1$ and signal $s2$ -> [ 3.2 ms ... 5.7 ms ] between events
```

Listing 1.7. Few timing constraints of the timed causal graph in Fig. 4

The first extension detects recurring events in event logs. In practice these events are usually periodic notifications containing status reporting information but they could also be signals or commands. The second extension detects a generalization of a command-reply pattern by allowing notifications in between. Such patterns are atomic from the point of view of a client because it is blocked until the reply is received. We end the section with a discussion about exploiting domain knowledge to detect hidden dependencies between events present in a log. For e.g. an event in the initialization phase may have an impact on the possible events available in the operational phase.

Using Client-Server Context to Distinguish Recurring Events. Control systems often generate periodic events, for e.g. notifications about status information generated by a server. If we simply transform a log containing recurring events into a causal graph, we may end up with a model where almost every event is possible after a recurring event.

For instance, consider the example in Fig. 6. Here we have a trace containing a periodic notification $n1$. If we ignore $n1$, then we observe that there is a causal relation between client-initiated events, i.e. command c is followed by signal $s1$ and then signal $s2$. When we transform such a trace into a causal graph, the causal relations are lost since $n1$ can be followed by either c, $s1$, $s2$. As a result, we end up allowing too much behavior.

One way to address this problem is by exploiting the fact that client-initiated events (commands and signals) and server-initiated events (notifications) occur in *context* of each other, i.e. a sequence of client-initiated events occur in the context of the last server-initiated event and vice versa. The additional *context* information is easily captured by

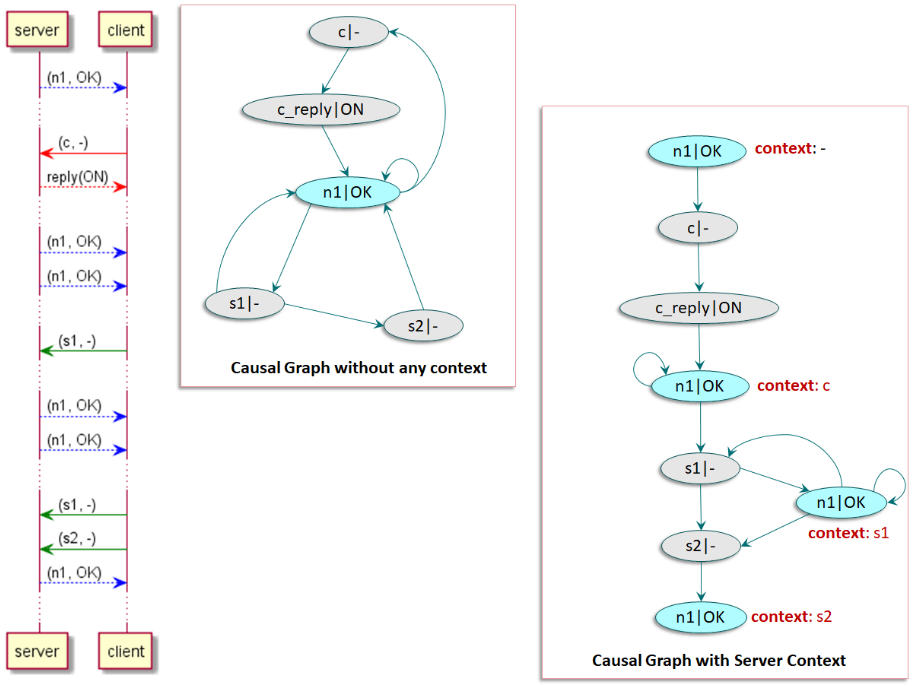


Fig. 6. Handling Periodic Events

extending the vertices of a causal graph with a context attribute. If a client or server initiated event is the first event of a trace then its context attribute is empty.

In the Fig. 6, we show how vertices representing periodic notification $n1$ are extended with context of client-initiated events. The occurrences of $n1$ are now distinguished based on the last occurring event from a client. As a result, causal relations between events $c, s1, s2$ are preserved.

Applying contexts to non-periodic events can have a negative effect on the final result. In practice, the user should be able to decide which events can benefit from a context attribute. This choice can be provided to the learner as part of its configuration parameters.

Inferring Compound Commands Many client-server based control systems support the possibility for a server to raise notifications during the execution of a command, i.e. a sequence of notifications before sending the corresponding reply. We refer to such a pattern as a *compound command*. Note that the client is blocked until it receives a reply from the server. In ComMA, we model such a pattern by adding one or more *notifications* to the body of a *transition trigger* referencing a *command*.

In the Listing 1.8, we give an example where a server can receive a command *switchOn* and as a response, it produces two notification *inventoryInfo* and *powerLevel* before returning a *reply* with value OK.

The trace induced by a *compound command* is a sequence of events starting with a *command*, ending with a *reply* and containing zero or more *notifications* in between, i.e.

```
transition trigger: switchOn
do: inventoryInfo(1)
    powerLevel(85)
    reply(OK)
next state: On
```

Listing 1.8. Compound Command

a sequence of the form $\langle c; n_1; n_2; \ldots n_m; c\_reply \rangle$, where $m \in \mathbb{N}$. In the Listing 1.9, we give an example.

```
Timestamp: 3.030400 Command: switchOn
Timestamp: 13.908800 Notification: inventoryInfo Parameter: integer : 1
Timestamp: 13.908800 Notification: powerLevel Parameter: integer : 85
Timestamp: 3.567788 Reply Parameter: Result::OK
```

Listing 1.9. Trace induced by Compound Command in Listing 1.8

Recall that due to restriction $R1$ in the previous section, we did not consider traces induced by a compound command. To relax this restriction, we make the following modifications to the concepts presented in the previous section.

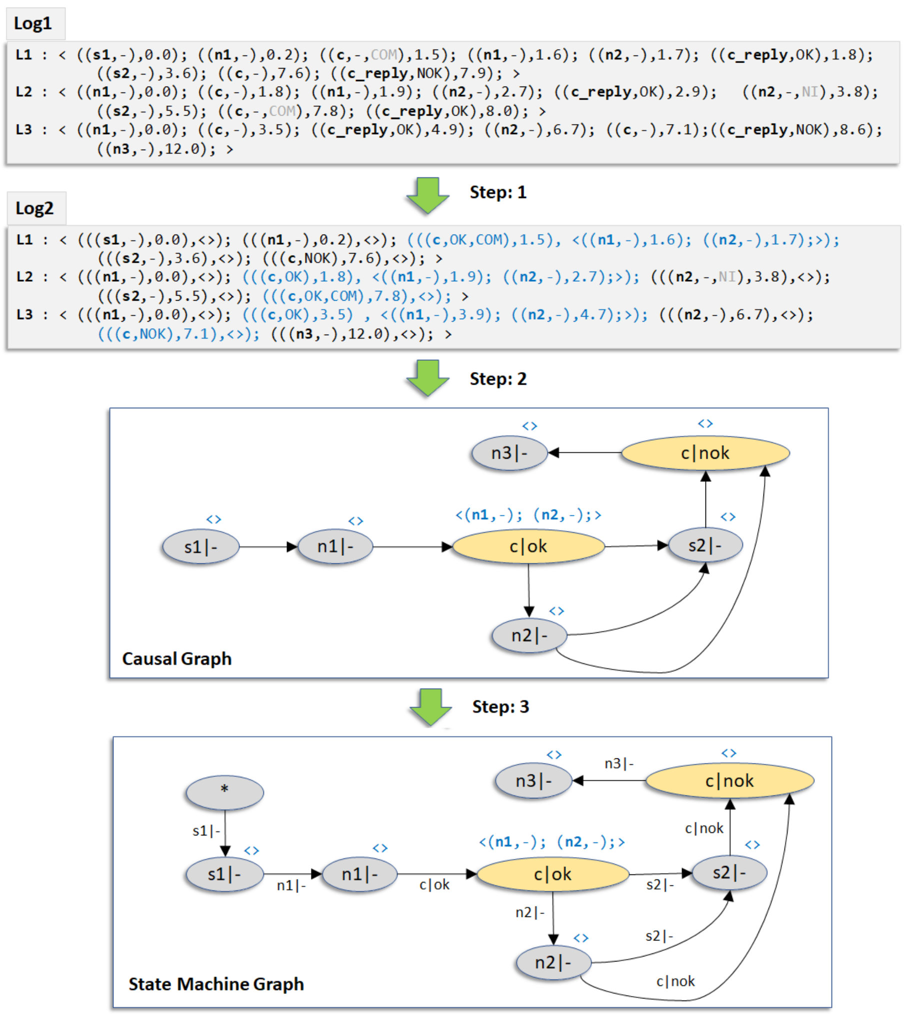
- Drop interface action type *REPLY* and relax the restriction on empty output strings for command events. So instead of using an explicit reply event, we use output string str of a *command* event (c, str) to capture its corresponding *reply* value.
- Lift the definition of an *event* to be the tuple (e, σ), where e is an *event* (as defined before) and σ is a sequence of *notification* events. We require that signal and notification events satisfy $\sigma = \epsilon$.

These modifications imply that event logs must be pre-processed to detect and aggregate subsequences induced by compound commands before constructing the causal graph with the newly extended notion of events. For instance the trace in Listing 1.9 is transformed into event $e = ((switchOn, \text{OK}), \langle \text{inventoryInfo}(5); \text{powerLevel}(85) \rangle)$. We give an example of pre-processing a log in step 1 of Fig. 7.

The transformation to a causal graph and then to a state machine stays the same (see step 2 and step 3 in Fig. 7). However, the algorithm to generate a ComMA state machine syntax must be modified to handle compound commands. This transformation is rather straightforward. For instance, event e must be transformed into the ComMA syntax presented in Listing 1.8.

Note that we still require each input trace to satisfy that for every *command* event there is a matching *reply* event.

Discussion It is often the case that we have some knowledge about the behavior of a component which would otherwise have required a large number of traces. There are two ways to capture such information.



Log1
```
L1 : < ((s1,-),0.0); ((n1,-),0.2); ((c,-,COM),1.5); ((n1,-),1.6); ((n2,-),1.7); ((c_reply,OK),1.8);
          ((s2,-),3.6); ((c,-),7.6); ((c_reply,NOK),7.9); >
L2 : < ((n1,-),0.0); ((c,-),1.8); ((n1,-),1.9); ((n2,-),2.7); ((c_reply,OK),2.9);   ((n2,-,NI),3.8);
          ((s2,-),5.5); ((c,-,COM),7.8); ((c_reply,OK),8.0); >
L3 : < ((n1,-),0.0); ((c,-),3.5); ((c_reply,OK),4.9); ((n2,-),6.7); ((c,-),7.1);((c_reply,NOK),8.6);
          ((n3,-),12.0); >
```

Log2
```
L1 : < (((s1,-),0.0),<>); (((n1,-),0.2),<>); (((c,OK,COM),1.5), <((n1,-),1.6); ((n2,-),1.7);;>);
          (((s2,-),3.6),<>); (((c,NOK),7.6),<>); >
L2 : < (((n1,-),0.0),<>); (((c,OK),1.8), <((n1,-),1.9); ((n2,-),2.7);;>); (((n2,-,NI),3.8),<>);
          (((s2,-),5.5),<>); (((c,OK,COM),7.8),<>); >
L3 : < (((n1,-),0.0),<>); (((c,OK),3.5) , <((n1,-),3.9); ((n2,-),4.7);;>); (((n2,-),6.7),<>);
          (((c,NOK),7.1),<>); (((n3,-),12.0),<>); >
```

Fig. 7. Inferring Compound Command in Traces

One way is to define a set of negative traces (i.e. forbidden order of events) and adapt the learning algorithm to take these orderings into account while constructing the causal graph. In practice such traces are not readily available and are time-consuming to create.

Another way is to specify domain knowledge in terms of temporal constraints [38] and use them in conjunction with a model checker to check for violations [45, 12]. The model checking process can be further augmented by formulating and asking questions to a user in terms of scenarios [16, 19] on the quality of the generalization. The user may accept or reject the generalization, or can provide a new temporal property that handles a larger class of scenarios. Constraints are usually specified as safety (must never be violated) and liveness (must eventually happen) properties [40].

As our algorithm takes into account only causal relations between pairs of directly follows events, a pattern such as event X is eventually followed by event Y is not exploited (long-term dependencies). For instance consider the interface model of IVendingMachine in List. 1.2. The interface state machine does not allow a vending machine with zero *items* to *switchOn* successfully (i.e. with a reply *OK*). It is hard to infer this behavior from traces only because other events can occur in between, e.g. *loadProduct* and *switchOn* (e.g. see List.1.4). Such a dependency can be expressed as a Linear Temporal Logic (LTL) formula.

6. Case Studies

In order to evaluate the learning algorithm we used two example cases for which we already constructed an interface manually earlier. For both interfaces there is a software implementation and a number of traces collected during testing the implementation. The first case is an interface of the power control unit; the second case is an interface of a third-party operating table.

The goal of the presented case studies is to evaluate the interface state machines generated by the learning algorithm in terms of size and understandability. We also compare the generated output to the original manually constructed state machines that were already present. Furthermore, we do not evaluate the generation of timing constraints because it is a work in progress.

Clearly the model inferred by the interface learner depends on the quality of the event log. Most of the times, event logs only contain only a subset of the possible events, usually determined by the execution context. Furthermore, only a part of the interface behavior may be represented by an event log. In order to characterize the input event log we measure the percentage of the used events and the coverage of the logs measured against the original interface models.

For both cases we first automatically checked for conformance of traces against the interface model and then applied the ComMA interface learner. The results from the power control unit case are shown in Table 1. The unit has 5 sub-interfaces: for inspecting the event log (logging); for inspecting the unit self-test results and software version (service); for updating the unit application software and configuration (utility); for monitoring startup and shutdown state (startup/shutdown); and for performing tests by injecting events (test interface).

Table 1. Results of learning experiment with power control unit

Interface	Nr.traces	Coverage	% used events	Size original interface	Size learned interface
Logging	1	83% transitions, 100% states	100%	1 machine, 2 states	4 states
Service	2	100% transitions and states	100%	1 machine, 1 state	5 states
Utility	2	7% transitions, 17% states	25%	1 machine, 6 states	4 states
Startup/shutdown	14	29% transitions, 71% states	54%	2 machines, 7 states	8 states
Test	1	16% transitions, 36% states	50%	1 machine, 11 states	3 states

For every interface the table columns indicate the number of traces used for monitoring and learning, the coverage of the trace set as a percentage of visited states and transitions in the original interface, the percentage of the used interface signature events, the total number of state machines and states in the original interface, and finally the number of states in the learned state machine.

The original Service interface is stateless (hence its specification has a single state). In the traces, the events were observed always in the same order (4 events in total), which explains why the learned state machine contains 5 states. It is anticipated that if the traces contain more permutations of the 4 events then some of the states can be merged by the learning algorithm.

For traces that cover only a small part of the behavior (demonstrated by low coverage and low percentage of used events) the number of states in the learned machine is lower than the original. This observation confirms the intuition that the learned behavior is a subset of the complete one.

All learned state machines are easy to understand partly due to the rules for forming the state names. Their size in terms of number of states is small reflecting the fact that the behavior of the interfaces in this case study is generally not complex.

The interface in the second case study (that of the operating table) is considerably more complex than the one for the power unit. The table can move along 5 axes indicated here as Axis 1 to 5. Moves on several axes can be executed in parallel. All moves have similar behavior captured in a simple state machine with 3 states. Thus the original interface specification consists of 5 orthogonal machines (one for each axis) plus one machine for the startup sequence and the generic parameter notifications. In addition, the table and its clients exchange keep-alive messages with high frequency which results in long traces. The keep-alive messages and parameter value notifications can happen in any state. The experimental set contains 6 traces: one with a move for each axis in isolation and one that combines moves along all axes. For simplicity, the traces do not include startup sequence.

Table 2. Results of learning experiment with the operating table

Trace	Coverage	Size learned behavior
Axis 1	15% transitions, 41% states	13 states
Axis 2	17% transitions, 47% states	13 states
Axis 3	18% transitions, 47% states	13 states
Axis 4	17% transitions, 41% states	13 states
Axis 5	30% transitions, 59% states	18 states
All axes	67% transitions, 94% states	31 states
All traces	71% transitions, 100% states	31 states

The results from this case study are summarized in Table 2. As in the previous case, for each trace set we indicate the coverage over the original interface. First, we applied the learner to one trace per axis to learn the behavior of each move in isolation (rows Axis 1-5). As can be seen, the trace coverage and the size of the results are comparable except for the trace for Axis 5 which appeared to contain moves along 2 axes. The row ”All axes” shows results for a trace containing moves along all axes. As a final step in the experiment we fed the learner will all the 6 traces together (last row). The total coverage of the input increases but the result is not very different in size and topology from the one obtained from the trace with all axes.

A machine with 31 states (obtained when all traces were used) is not easy to comprehend but we have to say that the original behavior specification is not simple either. It was difficult to identify the state behavior for a single move because the move-related events were interleaved with the keep-alive messages. We see a potential to reduce the size of this model by using domain knowledge to filter out keep-alive messages and some status update notifications that can happen at any time. The assumption is that these events do not have an effect on the other events and can therefore be isolated.

As a final observation we would like to note that the generated output with this new algorithm is generally smaller and more manageable than the one reported in the conference version of the paper.

The learned interfaces were inspected by the engineers who created the original specifications thus they had domain knowledge and were experienced in modeling the inspected behavior. As future work we plan to investigate interfaces without pre-existing specifications.

7. Related Work

Inferring state-based behavior from event logs is a well studied topic within Process Mining and Finite State Machine (FSM) Inference communities.

FSM Inference Approaches for FSM-based inference (grammar induction) are based on the learning framework described in [21] which shows that the class of regular languages cannot be identified in the limit from positive strings only, since this almost always leads to over generalization. For instance a self-loop model is always the simplest explanation for any given positive string but such a model is not very useful for analysis. So in practice, heuristics are used to control the amount of generalization present in the final model.

Most popular techniques for FSM inference are based on the *state merging* approach. They start by representing the set of available traces as a prefix tree acceptor and then in steps make generalizations by merging pairs of nodes based on an equivalence notion derived from the well-known Myhill-Nerode relation [36]. So the problem of inferring a state machine from a set of traces is reduced to identifying and scoring equivalent points in the traces that may be suitable merge candidates. Each merge produces a state machine with more allowed behavior (i.e. the set of all possible event orderings). Most popular strategies for generalizing prefix trees can be characterized by Bierman's K-tails algorithm [11] which works on the idea that two points in an execution trace (nodes representing states) are equivalent and can be merged if their future behaviors (up to k-steps) are identical. The work in [15] relaxes the equivalence notion to include subsets of possible behaviors, carried out in the context of discovering software engineering processes. The GK-tails method [33, 46] extends K-tails by considering the influence of data. The method relies on Daikon invariant detection to produce an extended FSM (EFSM), i.e. FSM with data guards on transitions.

When applying simple state merging algorithms to limited traces it is difficult to determine if a compatible merge is truly valid. A bad merge earlier on can have negative consequences on the end result. To some extent this issue is addressed by Evidence Driven State Merging (EDSM) approaches based on the Red-Blue Fringe framework [27]. In an EDSM based approach, each possible merge is given a score based on the amount of evidence of a good merge. The merge with the highest evidence is merged. An extension to the red-blue fringe state merging algorithm that takes into account timing information in traces and infers a timed automata from it, is presented in [43].

Some approaches rely on the user to determine if a merge is good or bad. The work in [16] presents strategies to formulate questions to the user. The user is also able to provide negative traces and temporal constraints to further improve the merge criteria. A similar approach based on model checking is presented in [45] which is extended in [12] using SAT solving techniques. The resulting Mealy machine is transformed into a non-deterministic Moore machine. The trivial solution for the case of positive traces only is the basis for the work in [39]. Note that this is different from the state merging approach.

A popular passive automata learning tool is Flexfringe [44] [7]. The tool provides an efficient implementation of the well-known evidence-driven blue-fringe state-merging algorithm and its probabilistic variants. There are many options to modify search strategies and the user can choose to extend functionality with custom algorithms or rely on standard algorithms such as RPNI [37], ALERGIA [14], EDSM [27], Overlap [23] etc. Another interesting tool is LearnLib [8]. The tool has a focus on active learning but there are also a few RPNI based passive learning algorithms. However most of these tools are difficult to use by a non-expert and solve only the general problem of inferring state machines. It is also not trivial to map the resulting models from these tools to domain specific concepts.

Comparison to the State Merging Approach To compare our approach to state merging approaches based on K-tails, we borrow a nice example of a text editor application from the work in [45]. The idea is simple: once a file is loaded, it can be edited. Only if a file has been edited then it may be saved. Finally a new document can only be loaded when the current document has been closed.

Recall that K-tails like other state merging approaches start by representing the set of available traces as a prefix tree acceptor. In each step, pairs of nodes of this tree are merged if their future behaviors (up to k-steps) are identical, and the resulting model is made deterministic. In addition to the k-value, the choice of pairs and evaluation of a merge rely on heuristics provided by the user.

In the Fig. 8, we present three traces from a log file of such an application and their corresponding FSM generated using the k-tails algorithm with values of $k = 1$ and $k = 2$ (note that the example log and generated state machines are borrowed from [45]).

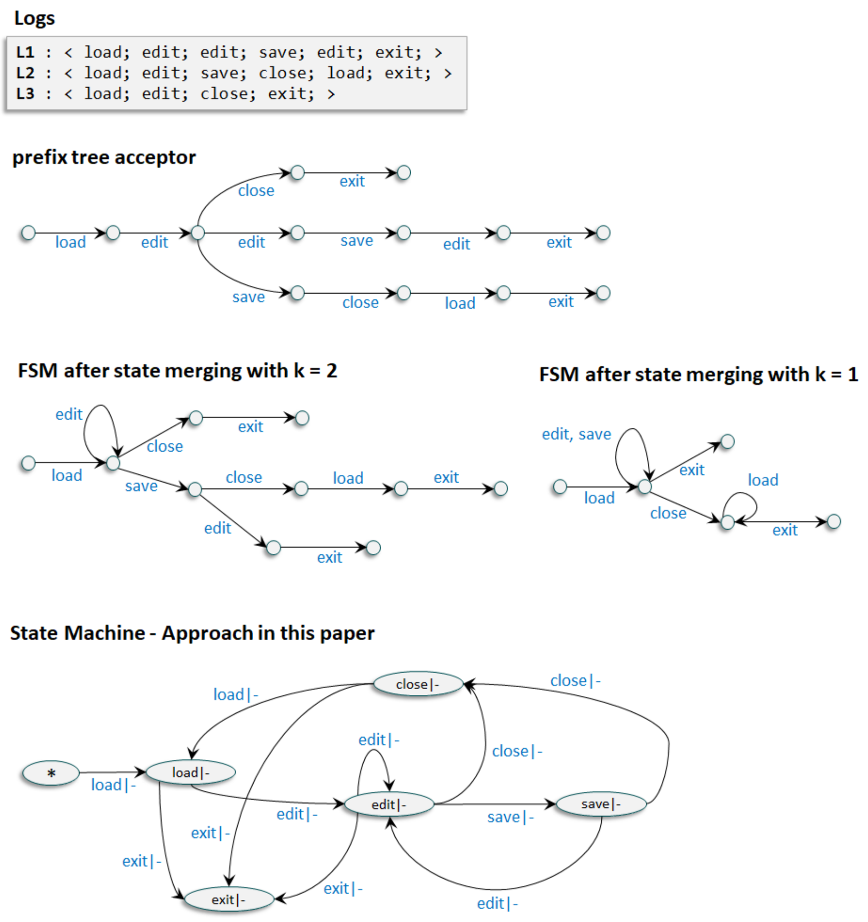
In general for a user it is difficult to determine the right value of k for which the resulting model is useful, since the user also has not much knowledge about the model being inferred.

Observe that for higher values of k, we get a FSM with less behavior (i.e. set of all possible event orders). So for $k = 2$ the FSM is almost the prefix tree, whereas for $k = 1$ we get a FSM with more behavior but

– it is possible to save without having edited the file
– it is possible to edit and save the first loaded file but not to the second loaded file.
– it is not possible to load a third file

In the same figure, we also show the state machine generated by our learning method. Since the model preserves the causal relations between events of a given trace, we do not suffer from the problems mentioned above. Furthermore, it is easy for the user to reason about the output state machine, since ordering relations can be checked very easily by either inspecting the set of input traces or its corresponding causal graph.

[7] https://automatonlearning.net/flexfringe/
[8] https://learnlib.de/


Logs

```
L1 : < load; edit; edit; save; edit; exit; >
L2 : < load; edit; save; close; load; exit; >
L3 : < load; edit; close; exit; >
```

Fig. 8. Comparing with the K-tails Approach

Process Mining The field of Process Mining aims to discover, monitor and improve processes by extracting knowledge from event logs [1]. Most commercially successful applications of process mining can be seen in the area of organizational business processes (Fluxicon, Celonis, UiPath, Process Gold, ProM, PM4PY)[9] [10]. Petri nets are a widely used formalism to model and analyse business processes [42]. So it is not surprising that most process mining techniques produce their output in terms of a Petri net [1]. In contrast to FSM based inference techniques, process mining techniques take into account the presence of concurrent behavior in event logs.

Early work on process mining can be traced back to three independent papers [5, 17, 15]. The work in [15] developed process discovery techniques in the context of software engineering processes. Among the three methods presented in this paper, the purely algo-

[9] https://www.celonis.com/, https://processgold.com/en/, https://www.fluxicon.com/
[10] https://www. my-invenio.com/, http://www.promtools.org/, https://pm4py.fit.fraunhofer.de/

rithmic approach (based on K-tails [11]) and Markovian approach to deal with noise were considered promising. Around the same time, the work in [5, 17] presented the first applications of process discovery in the context of business processes. The work in [17] adapts the K-tails algorithm with probabilistic elements. However none of the three approaches are able to discover concurrency.

The α-algorithm [2] was one of the first algorithms to mine concurrent behavior along side choices and causal dependencies. It is a simple technique that scans the event log for patterns and distinguishes them in log-based ordering relations, i.e. causal, choice and concurrency. These ordering relations are used by the algorithm to create places to connect transitions of the resulting Petri net.

The learning method presented in this paper uses ordering relations (like in the α-algorithm) between events of a log as its starting point. However for our case of inferring interface models, the presence of concurrent behavior in event logs is not yet a relevant aspect but gives us the nice possibility to extend our method to detect them in the future.

Another interesting approach concerns the theory of regions where the focus is on synthesizing a Petri net from a behavioral specification (for e.g. a transition system), such that the behavior is preserved. There are two main approaches, state-based region [20, 4, 18] and language-based region [10, 48]. Other approaches in process mining include frequency-based techniques such as the heuristics miner [47, 34], abstraction-based techniques such as the fuzzy miner [22] and genetic algorithms [3, 35] which take into account noise and incompleteness of event logs.

The quality of the discovered model with respect to the given log is measured using the four quality dimensions: fitness, simplicity, precision, and generalization [1]. Most approaches guarantee varying levels of fitness and re-discoverability. Among them region-based approaches achieve a good fitness. The problem of guaranteeing sound models with a good fitness is addresed in [30].

Most process mining approaches are not able to handle duplicate tasks since their occurrences are indistinguishable in an event log. As a result, models may become overly connected, negatively affecting the precision and simplicity of the model. Many solutions to detect duplicate tasks have been proposed [35, 32, 13] but the rules to identify them are not sufficiently general for all event logs. Note that we try to address this problem in our learning method by using context information(see Sec. 5).

Freely available tools [11] such as ProM and more recently a Python library (PM4PY) provide access to many mining algorithms. Similar to tools for FSM inference, these tools are also difficult to use by a non-expert and apply them to domain specific concepts. However, the many available professional process mining tools address these problems by providing tailored solutions for the domain of business process management.

8. Conclusions

We have presented a method to infer an interface state machine and a set of timing constraints from an event log. The inferred model is intended to serve as a starting point for subsequent modeling steps. In comparison to other approaches for passive learning, we

[11] http://www.promtools.org/, https://pm4py.fit.fraunhofer.de/

exploit client-server interaction patterns and also take into consideration data and timing information in event logs. Our method can also be configured to deal with recurring events, the choice of generated timing constraints and large data domains of parameters.

Like many process mining techniques [31], we also generate an intermediate causal graph using the directly-follows relation. For a user, such a graph serves as an intuitive way to visualize the information present in an event log and to reason about the resulting interface model. In contrast, the state machines produced by state merging approaches (see Sec. 7 and 4) are difficult for a user to reason about solely based on merge heuristics. Moreover having meaningful state names in interface models greatly improves readability which is also an important aspect for adoption.

Most parts of the method presented in this paper are available in ComMA[12], except support for compound commands and the generation of timing constraints. In future releases, we intend to add these missing features. As future work we see two interesting extensions of our method (1) Extensions to the concept of compound commands to capture more frequently occurring domain specific patterns such as cancelations, etc. [7, 9, 8], and (2) Extensions to infer behavior of components. In typical component-based systems, a component has a set of provided interfaces (to provide services) and a set of required interfaces (to consume services). So the goal is to infer a set of constraints between events of these interfaces based on evidence in event logs. Such an inference must be able to detect concurrent behavior in event logs. In this case exploiting the ordering relation for parallel tasks as presented in the α algorithm [2] could be a nice extension to our method.

Acknowledgments. The research is carried out as part of the VIVACE project under the responsibility of ESI (TNO) with Philips, Best as the carrying industrial partner. The VIVACE research is supported by the Netherlands Organisation for Applied Scientific Research TNO.

References

1. van der Aalst, W.: Process Mining - Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
2. Van der Aalst, W., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering 16(9), 1128–1142 (2004)
3. Van der Aalst, W.M., De Medeiros, A.A., Weijters, A.: Genetic process mining. In: International conference on application and theory of petri nets. pp. 48–69. Springer (2005)
4. Van der Aalst, W.M., Rubin, V., Verbeek, H., van Dongen, B.F., Kindler, E., Günther, C.W.: Process mining: a two-step approach to balance between underfitting and overfitting. Software & Systems Modeling 9(1), 87 (2010)
5. Agrawal, R., Gunopulos, D., Leymann, F.: Mining process models from workflow logs. In: International Conference on Extending Database Technology. pp. 467–483. Springer (1998)
6. Aslam, K., Cleophas, L., Schiffelers, R., van den Brand, M.: Interface protocol inference to aid understanding legacy software components. Software and Systems Modeling pp. 1–22 (2020)
7. Bera, D.: Petri nets for modeling robots. Ph.D. thesis, Department of Mechanical Engineering (2014)

[12] http://comma.esi.nl/

8. Bera, D., van Hee, K.M., van Osch, M., van der Werf, J.M.E.M.: A component framework where port compatibility implies weak termination. In: Proceedings of the International Workshop on Petri Nets and Software Engineering, Newcastle upon Tyne, UK, June 20-21, 2011. CEUR Workshop Proceedings, vol. 723, pp. 152–166
9. Bera, D., van Hee, K.M., van der Werf, J.M.: Designing weakly terminating ros systems. In: International Conference on Application and Theory of Petri Nets and Concurrency. pp. 328–347. Springer (2012)
10. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process mining based on regions of languages. In: International Conference on Business Process Management. pp. 375–383. Springer (2007)
11. Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behavior. IEEE transactions on Computers 100(6), 592–597 (1972)
12. Buzhinsky, I., Vyatkin, V.: Automatic inference of finite-state plant models from traces and temporal properties. IEEE Transactions on Industrial Informatics 13(4), 1521–1530 (2017)
13. Carmona, J., Cortadella, J., Kishinevsky, M.: A region-based algorithm for discovering petri nets from event logs. In: International Conference on Business Process Management. pp. 358–373. Springer (2008)
14. Carrasco, R.C., Oncina, J.: Learning stochastic regular grammars by means of a state merging method. In: International Colloquium on Grammatical Inference. pp. 139–152. Springer (1994)
15. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM Transactions on Software Engineering and Methodology (TOSEM) 7(3), 215–249 (1998)
16. Damas, C., Lambeau, B., Dupont, P., Van Lamsweerde, A.: Generating annotated behavior models from end-user scenarios. IEEE Transactions on Software Engineering 31(12), 1056–1073 (2005)
17. Datta, A.: Automating the discovery of as-is business process models: Probabilistic and algorithmic approaches. Information Systems Research 9(3), 275–301 (1998)
18. van Dongen, B.F., Busi, N., Pinna, G., van der Aalst, W.: An iterative algorithm for applying the theory of regions in process mining. In: Proceedings of the workshop on formal approaches to business processes and web services (FABPWS'07). pp. 36–55. Publishing House of University of Podlasie, Siedlce, Poland (2007)
19. Dupont, P., Lambeau, B., Damas, C., Lamsweerde, A.v.: The qsm algorithm and its application to software behavior model induction. Applied Artificial Intelligence 22(1-2), 77–115 (2008)
20. Ehrenfeucht, A., Rozenberg, G.: Partial (set) 2-structures. Acta Informatica 27(4), 343–368 (1990)
21. Gold, E.M.: Language identification in the limit. Information and control 10(5), 447–474 (1967)
22. Günther, C.W., Van Der Aalst, W.M.: Fuzzy mining–adaptive process simplification based on multi-perspective metrics. In: International conference on business process management. pp. 328–343. Springer (2007)
23. Heule, M.J., Verwer, S.: Software model synthesis using satisfiability solvers. Empirical Software Engineering 18(4), 825–856 (2013)
24. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation. Acm Sigact News 32(1), 60–65 (2001)
25. Kurtev, I., Hooman, J., Schuts, M.: Runtime monitoring based on interface specifications. In: ModelEd, TestEd, TrustEd. pp. 335–356. Springer (2017)
26. Kurtev, I., Schuts, M., Hooman, J., Swagerman, D.J.: Integrating interface modeling and analysis in an industrial setting. In: MODELSWARD. pp. 345–352 (2017)
27. Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In: International Colloquium on Grammatical Inference. pp. 1–12. Springer (1998)
28. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. International journal on software tools for technology transfer 1(1-2), 134–152 (1997)

29. Leemans, M., van der Aalst, W.M., van den Brand, M.G., Schiffelers, R.R., Lensink, L.: Software process analysis methodology–a methodology based on lessons learned in embracing legacy software. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 665–674. IEEE (2018)
30. Leemans, S.J., Fahland, D., van der Aalst, W.M.: Discovering block-structured process models from event logs-a constructive approach. In: International conference on applications and theory of Petri nets and concurrency. pp. 311–329. Springer (2013)
31. Leemans, S.J., Poppe, E., Wynn, M.T.: Directly follows-based process mining: Exploration & a case study. In: 2019 International Conference on Process Mining. pp. 25–32. IEEE (2019)
32. Li, J., Liu, D., Yang, B.: Process mining: Extending α-algorithm to mine duplicate tasks in process logs. In: Advances in Web and Network Technologies, and Information Management, pp. 396–407. Springer (2007)
33. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proceedings of the 30th international conference on Software engineering. pp. 501–510 (2008)
34. de Medeiros, A.K.A., van der Aalst, W.M., Weijters, A.: Workflow mining: Current status and future directions. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". pp. 389–406. Springer (2003)
35. de Medeiros, A.K.A., Weijters, A.J., van der Aalst, W.M.: Genetic process mining: an experimental evaluation. Data Mining and Knowledge Discovery 14(2), 245–304 (2007)
36. Nerode, A.: Linear automaton transformations. Proceedings of the American Mathematical Society 9(4), 541–544 (1958)
37. Oncina, J., Garcia, P.: Identifying regular languages in polynomial time. In: Advances in structural and syntactic pattern recognition, pp. 99–108. World Scientific (1992)
38. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). pp. 46–57. IEEE (1977)
39. Schuts, M., Hooman, J., Kurtev, I., Swagerman, D.J.: Reverse engineering of legacy software interfaces to a model-based approach. In: 2018 Federated Conference on Computer Science and Information Systems (FedCSIS). pp. 867–876. IEEE (2018)
40. Sistla, A.P.: Safety, liveness and fairness in temporal logic. Formal Aspects of Computing 6(5), 495–511 (1994)
41. Vaandrager, F.: Model learning. Commun. ACM 60(2), 86–95 (2017)
42. Verbeek, H.M., van der Aalst, W.M.: Analyzing bpel processes using petri nets. In: Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management. pp. 59–78 (2005)
43. Verwer, S., De Weerdt, M., Witteveen, C.: An algorithm for learning real-time automata. In: Benelearn 2007: Proceedings of the Annual Machine Learning Conference of Belgium and the Netherlands, Amsterdam, The Netherlands, 14-15 May 2007 (2007)
44. Verwer, S., Hammerschmidt, C.A.: flexfringe: a passive automaton learning package. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 638–642. IEEE (2017)
45. Walkinshaw, N., Bogdanov, K.: Inferring finite-state models with temporal constraints. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. pp. 248–257. IEEE (2008)
46. Walkinshaw, N., Taylor, R., Derrick, J.: Inferring extended finite state machine models from software executions. Empirical Software Engineering 21(3), 811–853 (2016)
47. Weijters, A., van Der Aalst, W.M., De Medeiros, A.A.: Process mining with the heuristics miner-algorithm. Technische Universiteit Eindhoven, Tech. Rep. WP 166, 1–34 (2006)
48. Van der Werf, J.M.E., van Dongen, B.F., Hurkens, C.A., Serebrenik, A.: Process discovery using integer linear programming. In: International conference on applications and theory of petri nets. pp. 368–387. Springer (2008)

49. Yang, N., Aslam, K., Schiffelers, R., Lensink, L., Hendriks, D., Cleophas, L., Serebrenik, A.: Improving model inference in industry by combining active and passive learning. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 253–263. IEEE (2019)

Debjyoti Bera received his MSc and PhD degrees from TU Eindhoven. He is currently a researcher at TNO-ESI, an innovation research center with strong partnerships with industry-leading high-tech companies. His research interests include model inference and applications of formal methods in industry.

Mathijs Schuts holds a PhD degree in computer science from the Radboud University Nijmagen. He works at Philips as a software designer. His main research interests include applying formal techniques and domain-specific languages in industry.

Jozef Hooman is a senior research fellow at ESI (TNO), an innovation research center with strong partnerships with industry-leading high-tech companies. In addition, Jozef is a full professor at the Radboud University Nijmegen on model-based development of embedded software.

Ivan Kurtev holds a PhD degree in computer science from University of Twente, the Netherlands. Currently he works as a modeling technologies expert in Altran Netherlands and is a part-time associate professor in the Eindhoven University of Technologies. His main interests are in the area of Model Based Engineering with a focus on applying domain-specific languages for efficient software development.

Received: May 20, 2020; Accepted: December 27, 2020.