Article:
Shin, D., Bianculli, D. and Briand, L. (2022) PRINS : scalable model inference for component-based system logs. Empirical Software Engineering, 27 (4). 87. ISSN: 1382-3256

https://doi.org/10.1007/s10664-021-10111-4

Empirical Software Engineering (2022) 27: 87
https://doi.org/10.1007/s10664-021-10111-4

PRINS: scalable model inference for component-based system logs

Donghwan Shin[1] · **Domenico Bianculli**[1] · **Lionel Briand**[1,2]

Accepted: 20 December 2021

Abstract

Behavioral software models play a key role in many software engineering tasks; unfortunately, these models either are not available during software development or, if available, quickly become outdated as implementations evolve. Model inference techniques have been proposed as a viable solution to extract finite state models from execution logs. However, existing techniques do not scale well when processing very large logs that can be commonly found in practice. In this paper, we address the scalability problem of inferring the model of a component-based system from large system logs, without requiring any extra information. Our model inference technique, called *PRINS*, follows a divide-and-conquer approach. The idea is to first infer a model of each system component from the corresponding logs; then, the individual component models are merged together taking into account the flow of events across components, as reflected in the logs. We evaluated *PRINS* in terms of scalability and accuracy, using nine datasets composed of logs extracted from publicly available benchmarks and a personal computer running desktop business applications. The results show that *PRINS* can process large logs much faster than a publicly available and well-known state-of-the-art tool, without significantly compromising the accuracy of inferred models.

Keywords Logs · Model inference · Component-based system

Communicated by: David Lo

This work has received funding from the Luxembourg National Research Fund (FNR) under grant agreement No C-PPP17/IS/11602677 and from the NSERC Discovery and Canada Research Chair programmes.

✉ Donghwan Shin
donghwan.shin@uni.lu

Domenico Bianculli
domenico.bianculli@uni.lu

Lionel Briand
lionel.briand@uni.lu

[1] University of Luxembourg, Esch-sur-Alzette, Luxembourg

[2] University of Ottawa, Ottawa, ON, Canada

1 Introduction

Behavior models of software system components play a key role in many software engineering tasks, such as program comprehension (Cook and Wolf 1998), test case generation (Fraser and Walkinshaw 2012), and model checking (Clarke et al. 2018). Unfortunately, such models are either scarce during software development or, if available, quickly become outdated as the corresponding implementations evolve, because of the time and cost involved in generating and maintaining them (Walkinshaw et al. 2010).

One possible way to overcome the lack of software models is to use *model inference* techniques, which extract models—typically in the form of Finite State Machine (FSM)—from execution logs. Although the problem of inferring a minimal FSM is NP-complete (Gold 1967), there have been several proposals of polynomial-time approximation algorithms to infer FSMs (Biermann and Feldman 1972; Beschastnikh et al. 2011) or richer variants, such as gFSM (guarded FSM) (Walkinshaw et al. 2016; Mariani et al. 2017) and gFSM extended with transition probabilities (Emam and Miller 2018), to obtain relatively faithful models.

Though the aforementioned model inference techniques are fast and accurate enough for relatively small programs, all of them suffer from scalability issues, due to the intrinsic computational complexity of the problem. This leads to out-of-memory errors or extremely long, impractical execution time when processing very large logs (Wang et al. 2015) that can be commonly found in practice. A recent proposal (Luo et al. 2017) addresses scalability using a distributed FSM inference approach based on MapReduce. However, this approach requires to encode the data to be exchanged between mappers and reducers in the form of key-value pairs. Such encoding is application-specific; hence, it cannot be used in contexts—like the one in which this work has been performed—in which the system is treated as a black-box (i.e., the source code is not available), with limited information about the data recorded in the system logs.

In this paper, we address the scalability problem of inferring a system model from the logs recorded during the execution (possibly multiple executions) of a system composed of multiple "components" (hereafter called *component-based system*), without requiring any extra information other than the logs. In this paper, we use the term "component" in a broad sense: the large majority of modern software systems are composed of different types of "components", such as modules, classes, and services; in all cases, the resulting system decomposition provides a high degree of modularity and separation of concerns. Our goal is to efficiently infer a system model that captures not only the components' behaviors but also the flow of events across the components as reflected in the logs.

Our approach, called *PRINS*, follows a *divide-and-conquer* strategy: we first infer a model of each component from the corresponding logs using a state-of-the-art model inference technique, and then we "stitch" (i.e., we do a peculiar type of merge) the individual component models into a system-level model by taking into account the interactions among the components, *as reflected in the logs*. The rationale behind this idea is that, though existing model inference techniques cannot deal with the size of all combined component logs, they can still be used to infer the models of individual components, since their logs tend to be sufficiently small. In other words, *PRINS* alleviates the scalability issues of existing model inference techniques by limiting their application to the smaller scope defined by component-level logs.

We implemented *PRINS* in a prototype tool, which internally uses MINT (Walkinshaw et al. 2016), the only publicly available state-of-the-art technique for inferring gFSMs, to infer the individual component models. We evaluate the scalability (in terms of execution time) and the accuracy (in terms of recall and specificity) of *PRINS* in comparison with

MINT (to directly infer system models from system logs), on nine datasets composed of logs extracted from publicly available benchmarks (He et al. 2020) and a personal computer (PC) running desktop business applications on a daily basis. The results show that *PRINS* is significantly more scalable than MINT and can even enable model inference when MINT leads to out-of-memory failures. It also achieves higher specificity than MINT (with a difference ranging between -3.1 pp and +34.9 pp, with pp=percentage points) while achieving lower recall than MINT (with a difference ranging between -23.5 pp and +0 pp). Through a detailed analysis, we determined that a lower recall for *PRINS* only happens when logs are inadequate to infer accurate models, using any of the techniques. We also propose a simple and practical metric for engineers to easily predict (and thus improve) such cases before running model inference. With adequate logs, *PRINS* therefore provides a comparable or even better accuracy.

To summarize, the main contributions of this paper are:

- the *PRINS* approach for taming the scalability problem of inferring the model of a component-based system from the individual component-level logs, when no additional information is available;
- the novel *stitching* algorithm that "combine" individual component models together taking into account the flow of events across components as recorded in logs;
- a publicly available implementation of *PRINS* (see Section 5.8);
- the empirical evaluation, in terms of scalability and accuracy, of *PRINS* and its comparison with the state-of-the-art model inference tool.

The rest of the paper is organized as follows. Section 2 gives the basic definitions of logs and models that will be used throughout the paper. Section 3 illustrates the motivating example. Section 4 describes the different stages of *PRINS*. Section 5 reports on the evaluation of *PRINS*. Section 6 discusses related work. Section 7 concludes the paper and provides directions for future work.

2 Background

This section provides the basic definitions for the main concepts that will be used throughout the paper.

2.1 Logs

A *log* is a sequence of log entries; a *log entry* contains a timestamp (recording the time at which the logged event occurred), a component (representing the name of the component where the event occurred), and a log message (with run-time information related to the logged event). A log message is typically a block of free-form text that can be further decomposed (Zhu et al. 2019; He et al. 2017; Messaoudi et al. 2018; El-Masri et al. 2020) into an event template, characterizing the event type, and the parameter values of the event, which are determined at run time. For example, given the log entry "`15:37:56 - Master - end (status=ok)`", we can see that the event `end` of the component `Master` occurred at timestamp `15:37:56` with the value `ok` for parameter `status`.

More formally, let C be the set of all components of a system, ET be the set of all events that can occur in the system, V be the set of all mappings from event parameters to their concrete values, for all events $et \in ET$, and L be the set of all logs retrieved for the system; a log $l \in L$ is a sequence of log entries $\langle e_1, e_2, \ldots, e_n \rangle$, with $e_i = (ts_x, cm_i, et_i, v_i)$, $ts_i \in \mathbb{N}$,

$cm_i \in C$, $et_i \in ET$, and v_i is a vector of parameter values over V. To denote individual log entries, we use the notation $e_{i,j}$ for the i-th log entry in the j-th execution log.

2.2 Models

In this paper, we represent the models inferred for a system for a component as guarded Finite State Machines (gFSMs). Informally, a gFSM is an "extended" finite state machine whose transitions are triggered by the occurrence of an event and are guarded by a function that evaluates the values of the event parameters.

More formally, let ET and V be defined as above. A gFSM is a tuple $m = (S, ET, G, \delta, s_0, F)$, where S is a finite set of states, G is a finite set of guard functions of the form $g : V \rightarrow \{True, False\}$, δ is the transition relation $\delta \subseteq S \times ET \times G \times S$, $s_0 \in S$ is the initial state, and $F \subseteq S$ is the set of final states. A gFSM m makes a guarded transition from a (source) state $s \in S$ to a (target) state $s' \in S$ when reading an input log entry $e = (ts, cm, et, v)$, written as $s \xrightarrow{e} s'$, if $(s, et, g, s') \in \delta$ and $g(v) = True$. We say that a guarded transition is *deterministic* if there is at most one target state for the same source state and the same log entry. Otherwise, it is *non-deterministic*. Based on this, we say that a gFSM is deterministic if all of its guarded transitions are deterministic; otherwise, the gFSM is non-deterministic. We say that a gFSM m *accepts* a log $l = \langle e_1, \ldots, e_n \rangle$ if there exists a sequence of states $\langle \gamma_0, \ldots, \gamma_n \rangle$ such that (1) $\gamma_i \in S$ for $i = 0, \ldots, n$, (2) $\gamma_0 = s_0$, (3) $\gamma_{i-1} \xrightarrow{e_i} \gamma_i$ for $i = 1, \ldots, n$, and (4) $\gamma_n \in F$.

3 Motivating Example

This section presents a simple example to motivate and demonstrate our work.

Let us consider an imaginary system composed of two components, `Master` and `Job`; Fig. 1 depicts the set of logs $L_S = \{l_1, l_2\}$ recorded during the executions of the system. Entries in the logs are denoted using the notation introduced in Section 2.1; for instance, log entry $e_{8,1}$ corresponds to the tuple $(\cdot, \texttt{Master}, end, [ok])$, where the event is "*end*" and the value for its first (and only) parameter is "*ok*". Notice that in Fig. 1 we use a short form (as in "*end (ok)*") to indicate both an event and its parameter value; also, we omit timestamps in the running example logs as they are not used in our approach.

First execution log l_1			Second execution log l_2		
ID	Component	Event	ID	Component	Event
$e_{1,1}$	`Master`	*start*	$e_{1,2}$	`Master`	*start*
$e_{2,1}$	`Job`	*init*	$e_{2,2}$	`Job`	*init*
$e_{3,1}$	`Master`	*working*	$e_{3,2}$	`Master`	*working*
$e_{4,1}$	`Job`	*try*	$e_{4,2}$	`Job`	*try*
$e_{5,1}$	`Job`	*pass*	$e_{5,2}$	`Job`	*wait*
$e_{6,1}$	`Job`	*try*	$e_{6,2}$	`Job`	*wait*
$e_{7,1}$	`Job`	*pass*	$e_{7,2}$	`Job`	*fail*
$e_{8,1}$	`Master`	*end (ok)*	$e_{8,2}$	`Master`	*end (err)*

Fig. 1 Running example logs $L_S = \{l_1, l_2\}$, inspired by Hadoop logs (He et al. 2020)

A software engineer is tasked with building a finite-state model of the system that accurately captures the behavior observed in the logs. However, the engineer cannot rely on the system source code since it is not available. This is the case, for example, where the system is mainly composed of heterogeneous, 3rd-party components for which neither the source code nor the documentation are available (Aghajani et al. 2019; Palmer and McAddis 2019; Rios et al. 2020). The only information about the system to which engineers have access is represented by the execution logs L_S. To perform this task, the engineer uses a tool implementing one of the state-of-the-art *model inference techniques* (Walkinshaw et al. 2016; Mariani et al. 2017; Emam and Miller 2018) proposed in the literature; the tool takes as input the logs L_S and returns the system model m_S shown in Fig. 2a. Intuitively, we can see that m_S properly reflects the flow of events recorded in L_S. However, when the engineer tries to execute the model inference tool on much larger logs of the same system, she observes that the tool does not terminate within a practical time limit (e.g., one day). Indeed, due to the intrinsic complexity of the model inference problem (Gold 1967), the time complexity of state-of-the-art model inference algorithms is polynomial (Lang et al. 1998; Emam and Miller 2018) in the size of the input logs.

To address this problem, the engineer decides to use our new approach, *PRINS*: it takes as input the logs L_S in Fig. 1 and returns the *same* system model m_S shown in Fig. 2a; the main difference with the tool used in the previous attempts is that *PRINS* takes considerably less time to yield a system model.

The main idea behind *PRINS* is to tackle the intrinsic complexity of model inference by means of a *divide-and-conquer* approach: *PRINS* uses existing model inference techniques to infer a model, not for the whole system but *for each component*. Figure 2b and c show the component models m_M and m_J for the `Master` and `Job` components, respectively. Component-level model inference is one of the main contributors to the significant reduction of the execution time achieved by *PRINS*. Furthermore, component-level model inference can be easily parallelized.

However, before yielding a system model, *PRINS* needs to properly "combine" the individual component models. In our running example, this means building the m_S model shown

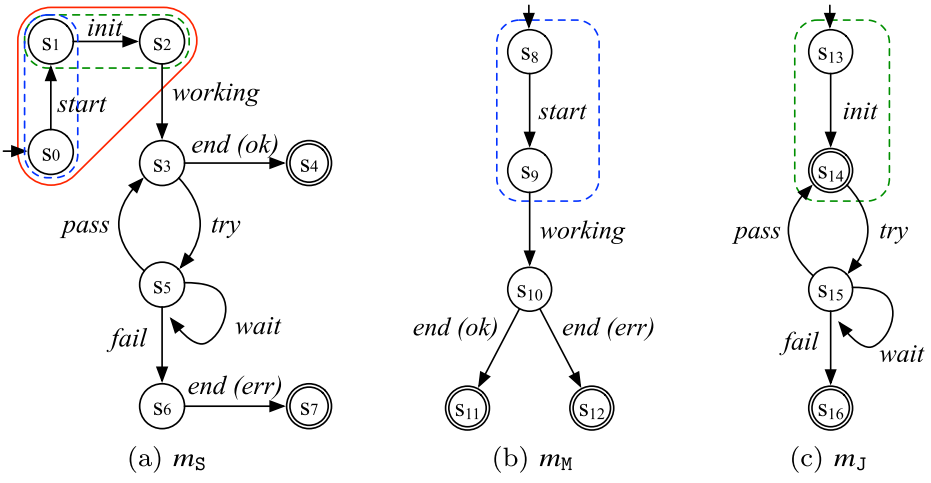

Fig. 2 Models corresponding to the running example logs (m_S: system model, m_M: model for component `Master`, m_J: model for component `Job`)

in Fig. 2 by "combining" the component models in $M_C = \{m_{\texttt{M}}, m_{\texttt{J}}\}$ and shown in Fig. 2b and c. This is a challenging problem: we cannot simply concatenate or append the two component models together, because the result would not conform to the flow of events across the components recorded in the logs. In our running example logs, it is recorded that the event *start* of `Master` is immediately followed by the event *init* of `Job`. Such a flow of events recorded in the logs should be represented in the final system model produced by *PRINS*. To efficiently and effectively solve this problem, we propose a novel algorithm for *stitching* component models in the context of model inference.

4 Scalable Model Inference

Our technique for scalable model inference follows a *divide-and-conquer* approach. The main idea is to first *infer* a model of each system component from the corresponding logs that are generated by the *projection* of system logs on the components; then, the individual component models are *stitched* together taking into account the flows of the events across the components, *as reflected in the logs*. We call this approach *PRINS* (*PR*ojection-*IN*ference-*S*titching). The rationale behind *PRINS* is that, though existing (log-based) model inference techniques cannot deal with the size of system logs, they can still be used to accurately infer the models of individual components, since their logs are sufficiently small for the existing model inference techniques to work. As anticipated in Section 3, the challenge is then how to "stitch" together the models of the individual components to build a system model that reflects not only the components' behaviors but also the flow of events across the components, while preserving the accuracy of the component models. Tackling this challenge is our main contribution, as detailed in Section 4.3.

Figure 3 outlines the workflow of *PRINS*. It takes as input the logs of the system under analysis, possibly coming from multiple executions; it returns a system model in the form of a gFSM. *PRINS* is composed of four main stages: *projection*, *inference*, *stitching*, and *determinization*. The projection stage produces a set of logs for each component from the input system logs. The component logs are then used to infer individual component models in the inference stage. The stitching stage combines the component models into a non-deterministic system model. Last, the determinization stage transforms the non-deterministic model into a deterministic model that is the output of *PRINS*. The four stages are described in detail in the following subsections.

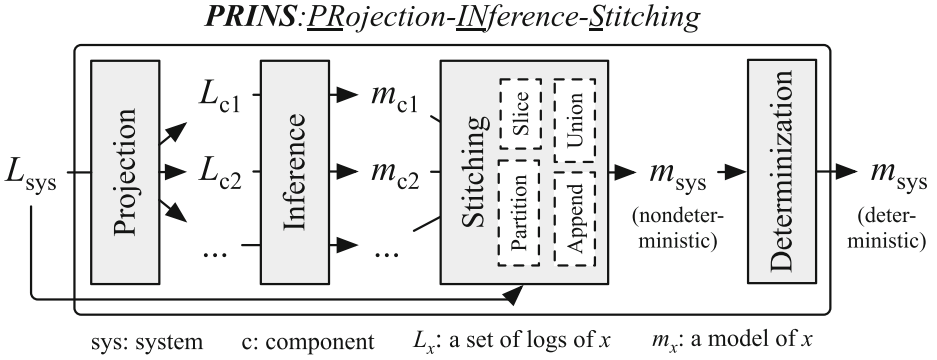

Fig. 3 Overview of *PRINS*

We remark that *PRINS* does not require any extra information (e.g., source code and documentation) other than logs. Furthermore, we do not restrict logs to be produced by one thread/process: as one can see in the replication package for our evaluation (see Section 5.8), individual logs for many of our subject systems are already produced by multiple threads/processes (distinguished by tid/pid). Our only assumption is that log entries contain the name of the "component" that generated them. This assumption is realistic since this is common in practice, as shown in real logs (He et al. 2020). Also, as indicated in Section 1, we use the term "component" in a broad sense (e.g., modules, classes) to represent an architectural "part" of a system. Therefore, *PRINS* is applicable to any software system composed of multiple components as long as their behavior is recorded in logs.

4.1 Projection

This stage generates a set of component logs—which will be used to infer a model for each component—from system logs. For instance, for our running example logs $L_{\mathtt{S}} = \{l_1, l_2\}$ shown in Fig. 1, we want to generate the set of logs for the `Master` component $L_{\mathtt{M}} = \{\langle e_{1,1}, e_{3,1}, e_{8,1}\rangle, \langle e_{1,2}, e_{3,2}, e_{8,2}\rangle\}$, and the set of logs for the `Job` component $L_{\mathtt{J}} = \{\langle e_{2,1}, e_{4,1}, e_{5,1}, e_{6,1}, e_{7,1}\rangle, \langle e_{2,2}, e_{4,2}, e_{5,2}, e_{6,2}, e_{7,2}\rangle\}$. To achieve this, we define the *projection* operation as follows. Let L be a set of logs of a system and C be a set of components of the system; the projection of L for a component $c \in C$, denoted with $L|c$, is the set of logs obtained from L by removing all occurrences of log entries of all $c' \in C$ where $c' \neq c$. For the running example, we have $L_{\mathtt{S}}|\mathtt{Master} = L_{\mathtt{M}}$ and $L_{\mathtt{S}}|\mathtt{Job} = L_{\mathtt{J}}$.

4.2 Inference

This stage infers individual component models from the sets of component logs generated from the projection stage. This is straightforward because inferring a (component) model from a set of logs can be achieved using an off-the-shelf model inference technique. We remark that *PRINS* does not depend on any particular model inference technique, as long as it yields a deterministic FSM (or a deterministic gFSM[1]) as a resulting model. Also, *PRINS* can infer multiple component models in parallel because the inference processes of the individual component models are independent from each other. For the running example, using an off-the-shelf model inference tool like MINT (Walkinshaw 2018) on the logs in $L_{\mathtt{M}}$ and $L_{\mathtt{J}}$, we obtain models $m_{\mathtt{M}}$ (see Fig. 2b) and $m_{\mathtt{J}}$ (see Fig. 2c), respectively.

We want to note that the parallelization of component model inference is just a byproduct of using the divide-and-conquer approach enabled by the central component of *PRINS*: stitching (Section 4.3).

4.3 Stitching

Individual component models generated from the inference stage are used in this *stitching* stage, which is at the core of *PRINS*. In this stage, we build a system model that captures not only the components' behaviors inferred from the logs but also the flow of events across components as reflected in the logs. For the running example, this means building a model

[1] A deterministic gFSM $m = (S, ET, G, \delta, s_0, F)$ with $\delta : S \times ET \times G \to S$ can be easily converted into a deterministic FSM $m' = (S, \Sigma, \delta', s_0, F)$ with $\delta' : S \times \Sigma \to S$ where $\Sigma = ET \times G$.

that is as "similar" as possible to $m_{\mathtt{S}}$, using models $m_{\mathtt{M}}$ and $m_{\mathtt{J}}$ as well as the input logs $L_{\mathtt{S}}$ reflecting the flow of events.

The idea of stitching comes from two important observations on the system and component models: (1) A system model is the composition of *partial models* of the individual components; this means that *partial behaviors* of components are *interleaved* in a system model. (2) The component partial models (included within a system model) are combined together (i.e., appended) *according to the flow of events recorded in logs*, since the system model must be able to accept the logs that were used to infer it.

For example, in the models shown in Fig. 2, we can see that the subgraph of $m_{\mathtt{S}}$, enclosed with a red solid line, contains two partial models: one, called $m^p_{\mathtt{M}}$ and enclosed with a blue dashed line, extracted from $m_{\mathtt{M}}$ (including the states s_8 and s_9—mapped to s_0 and s_1 in $m_{\mathtt{S}}$—and the corresponding transition labeled with *start*) and the other, called $m^p_{\mathtt{J}}$ and enclosed with a green dashed line, extracted from $m_{\mathtt{J}}$ (including the states s_{13} and s_{14}—mapped to s_1 and s_2 in $m_{\mathtt{S}}$—and the corresponding transition labeled with *init*). Notice that the partial models $m^p_{\mathtt{M}}$ and $m^p_{\mathtt{J}}$ correspond to the partial (behaviors recorded in the) logs $\langle e_{1,1}\rangle$ and $\langle e_{2,1}\rangle$, respectively, that are determined by the interleaving of components in the system log l_1 shown in Fig. 1. Furthermore, in $m_{\mathtt{S}}$, $m^p_{\mathtt{J}}$ is appended to $m^p_{\mathtt{M}}$, reflecting the fact that event *start* (from component `Master`) is immediately followed by *init* (from component `Job`) in the input logs.

Based on these observations, we propose a novel stitching technique that first "*slices*" individual component models into partial models according to the component interleavings shown in logs; then it "*appends*" partial models according to the flow of the events recorded in logs. However, the behaviors of components recorded in logs can be different from execution to execution (for instance, see the difference in terms of recorded events between l_1 and l_2 in our running example). To address this, we first build an intermediate, system-level model *for each execution* (i.e., for each log) and then merge these models together at the end.

The ***Stitch*** algorithm (whose pseudocode is shown in Algorithm 1) takes as input a set of logs L_{sys} and a set of component models M_C; it returns a system model m_{sys} (built from the elements in M_C) that accepts L_{sys}.

Algorithm 1 ***Stitch***.

Input : Set of System Logs (Structured) L_{sys}
Set of Component Models M_C
Output: System Model m_{sys}

```
1  Set of Models A ← ∅
2  foreach Log l_sys ∈ L_sys do
3      Model m_a ← emptyModel
4      initializeSliceStartStates(M_C)
5      List of Logs P ← Partition(l_sys)
6      foreach Log l_c ∈ P do
7          Model m_c ← getCorrespondingModel(c, M_C)
8          Model m_sl ← Slice(m_c, l_c)
9          m_a ← Append(m_a, m_sl)
10     A.add(m_a)
11 Model m_sys ← Union(A)
12 return m_sys
```

The algorithm builds a system-level model m_a for each system log $l_{sys} \in L_{sys}$ (lines 2–10). To build m_a for a given l_{sys}, the algorithm first initializes m_a as an empty model (lin3) and initializes the start states of all components models in M_C to their initial states (line 4). The algorithm then partitions l_{sys} into a list of logs P, each one corresponding to log entries of one component, according to the component interleavings shown in l_{sys} (using algorithm ***Partition*** at line 5, described in detail in Section 4.3.1). For each log $l_c \in P$ (lines 6-9), the algorithm retrieves the component model $m_c \in M_C$ of the component c that produced l_c, slices it (using algorithm ***Slice*** at line 8, described in detail in Section 4.3.2) into a partial model m_{sl} that accepts *only* log l_c, and then appends m_{sl} to m_a (using algorithm ***Append*** at line 9, described in detail in Section 4.3.3). During the iteration over the system logs in L_{sys}, the resulting system-level models m_a are collected in the set A. Last, the models in A are combined into a single model m_{sys} (using algorithm ***Union*** at line 11, described in detail in Section 4.3.4). The algorithm ends by returning m_{sys} (line 12), inferred from all logs in L_{sys}.

Before illustrating an example for ***Stitch***, let us first present the details of the auxiliary algorithms ***Partition***, ***Slice***, ***Append***, and ***Union***.

4.3.1 Partition

This algorithm takes as input a system log l (i.e., a sequence of log entries from various components); it partitions l into a sequence of logs P, where each log $l_c \in P$ is the longest uninterrupted sequence of log entries produced by the same component, and returns P. By doing this, we can divide a system log into component-level logs, each of which represents the longest uninterrupted partial behavior for a component, while preserving the flow of events across components as recorded in the system log.

For instance, when the function takes as input the running example log l_1, it returns $P = \langle l_{c,1}, l_{c,2}, \ldots, l_{c,5} \rangle$ where $l_{c,1} = \langle e_{1,1} \rangle$, $l_{c,2} = \langle e_{2,1} \rangle$, $l_{c,3} = \langle e_{3,1} \rangle$, $l_{c,4} = \langle e_{4,1}, e_{5,1}, e_{6,1}, e_{7,1} \rangle$, and $l_{c,5} = \langle e_{8,1} \rangle$.

4.3.2 Slice

This algorithm (whose pseudocode is shown in Algorithm 2) takes as input a component model m_c and a component log l_c; it returns a new model m_{sl}, which is the sliced version of m_c and accepts only l_c.

Algorithm 2 *Slice*.

Input : Component Model m_c
Component Log $l_c = \langle e_1, e_2, \ldots \rangle$
Output: Component Model m_{sl}

```
1 Model m_sl ← emptyModel
2 State s ← getSliceStartState(m_c)
3 foreach Log Entry e ∈ l_c do
4     Guarded Transition gt ← getGT(m_c, s, e)
5     m_sl ← addGTAndStates(m_sl, gt)
6     s ← getTargetState(gt)
7 updateSliceStartState(m_c, s)
8 return m_sl
```

First, the algorithm retrieves the state of m_c that will become the initial state s of the sliced model m_{sl} (line 2). Upon the first invocation of ***Slice*** for a certain model m_c, s will be the initial state of m_c; for the subsequent invocations, s will be the last state visited in m_c when running the previous slice operations. Note that there is always only one last visited state because m_c is deterministic, as described in Section 4.2. Starting from s, the algorithm performs a run of m_c as if it were to accept l_c by iteratively reading each log entry $e \in l_c$: the traversed states and guarded transitions of m_c are added into m_{sl} (lines 3-6). After the end of the iteration, the algorithm records the last state s visited in m_c (line 7), which is the (only one) final state of m_{sl} and will be used as the initial state of the next slice on m_c. The algorithm ends by returning m_{sl}.

For example, let us consider the case where ***Slice*** is called with parameters $m_c = m_{\mathrm{M}}$ and $l_c = \langle e_{1,1} \rangle$, and the slice start state returned by *getSliceStartState* for m_{M} is the initial state s_8. Starting from $s = s_8$, a run of m_{M} is performed: reading log entry $e_{1,1}$ results in making the guarded transition to s_1 in m_{M}. This results in m_{sl} to include the guarded transition from s_8 to s_9 with label *start* as well as the states s_8 and s_9; the call to function *getTargetState* updates s to s_9. Since there is no more log entry in l_c, s_9 is the final state of m_{sl} and is the slice start state for the next call to ***Slice*** for m_{M}. The resulting m_{sl} is $slice_1$ shown in Fig. 4.

4.3.3 Append

This algorithm takes as input two models m_a and m_{sl}; it returns an updated version of m_a built by appending m_{sl} to the end of the original version of m_a. If m_a is an empty model (i.e., when ***Slice*** is called for the first time after the initialization of m_a in line 3 in Algorithm 1), the algorithm simply returns m_{sl}. Otherwise, the algorithm merges the final state of m_a and the initial state of m_{sl} and ends by returning the updated m_a. Merging two states s_x and s_y is done by simply changing both the source states of all outgoing transitions of s_y and the target states of all incoming transitions of s_y as s_x. Note that a sliced model m_{sl} has only one final state as noted in Section 4.3.2, and therefore so does m_a.

For example, let us consider the case where ***Append*** is called with parameters $m_a = slice_1$ and $m_{sl} = slice_2$ shown in the left block of Fig. 4. The algorithm merges s_9 (i.e.,the final state of m_a) and s_{13} (i.e., the initial state of m_{sl}), resulting in $m_{a'}$ shown in the right block of Fig. 4.

4.3.4 Union

This algorithm takes as input a set of models A; it returns a model m_u that is able to accept all logs that can be accepted by all models in A. To do this, the algorithm simply merges the initial states of all models in A, and ends by returning the merged model as m_u.

Fig. 4 Illustration of appending two sliced models generated by ***Slice*** for $l_{c,1} = \langle e_{1,1} \rangle$ and $l_{c,2} = \langle e_{2,1} \rangle$. They are appended by ***Append***, resulting in $m_{a'}$ that accepts $\langle e_{1,1}, e_{2,1} \rangle$

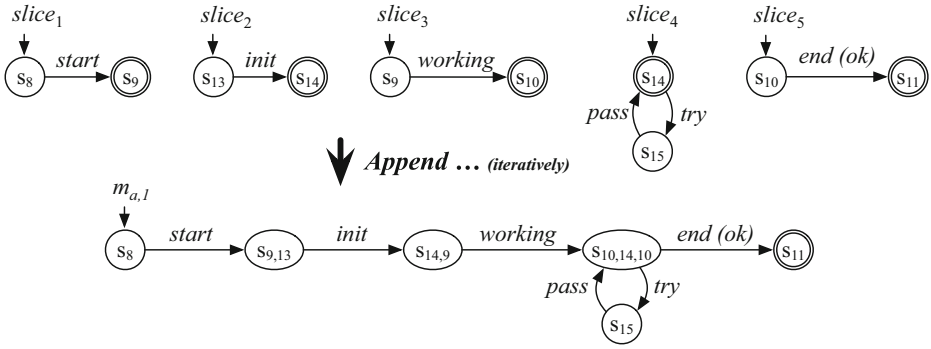

Fig. 5 Illustration of building a system-level model for the running example log l_1. The five sliced models are generated by ***Slice*** according to the partition of l_1. They are appended by ***Append*** to build a system-level model $m_{a,1}$ that accepts l_1

We remark that merging states in ***Append*** and ***Union*** can make the resulting model non-deterministic. Actually, the two algorithms are simplified[2] versions of the standard NFA (Non-deterministic Finite Automata) concatenation and union operation, respectively. We will discuss non-determinism later in the determinization stage (see Section 4.4).

4.3.5 Application of Stitch to the running example

Let us consider the case where the ***Stitch*** algorithm is called with parameters $L_{sys} = \{l_1, l_2\}$ and $M_C = \{m_{\text{M}}, m_{\text{J}}\}$. For l_1, the call to ***Partition*** yields $P = \langle l_{c,1}, l_{c,2}, \ldots, l_{c,5}\rangle$ where $l_{c,1} = \langle e_{1,1}\rangle$, $l_{c,2} = \langle e_{2,1}\rangle$, $l_{c,3} = \langle e_{3,1}\rangle$, $l_{c,4} = \langle e_{4,1}, e_{5,1}, e_{6,1}, e_{7,1}\rangle$, and $l_{c,5} = \langle e_{8,1}\rangle$. For each $l_{c,i} \in P$, the call to ***Slice*** yields a sliced model $slice_i$ shown in the top block of Fig. 5, originated from the component models m_{M} and m_{J} shown in Fig. 2. The five sliced models are appended to m_a using ***Append***, resulting in a system-level model $m_{a,1}$ shown at the bottom of Fig. 5. The state names of $m_{a,1}$ show how the initial and final states of the sliced models were merged. For example, $s_{10,14,10}$ is generated by merging s_{10} (i.e., the final state of $slice_3$), s_{14} (i.e., the initial and final state of $slice_4$), and s_{10} (i.e., the initial state of $slice_5$). Note that each $slice_i$ accepts the corresponding $l_{c,i} \in P$ and, as a result, $m_{a,1}$ accepts l_1. The algorithm ends the iteration for l_1 by adding $m_{a,1}$ into A and moves on to the next iteration to process log l_2. After this second iteration completes, the newly built model $m_{a,2}$ for l_2 is added to A; the call to ***Union*** yields a system model m_{uni}, shown at the top of Fig. 6. We can see that m_{uni} is composed of $m_{a,1}$ (i.e., the upper subgraph enclosed with a blue dashed line) and $m_{a,2}$ (i.e., the lower subgraph enclosed with a red dashed line).

In the above example, we can see that the output model m_{uni} accepts the input logs L_{sys} as expected. However, m_{uni} is actually not equivalent to m_{S} shown in Fig. 2a: there exist potential logs that only m_{S} can accept, but m_{uni} cannot. We will see how m_{uni} can be further

[2]To be precise, merging two states is not equivalent to introducing an epsilon-transition from one to another, but equivalent to introducing bi-directional epsilon-transitions between the two states.

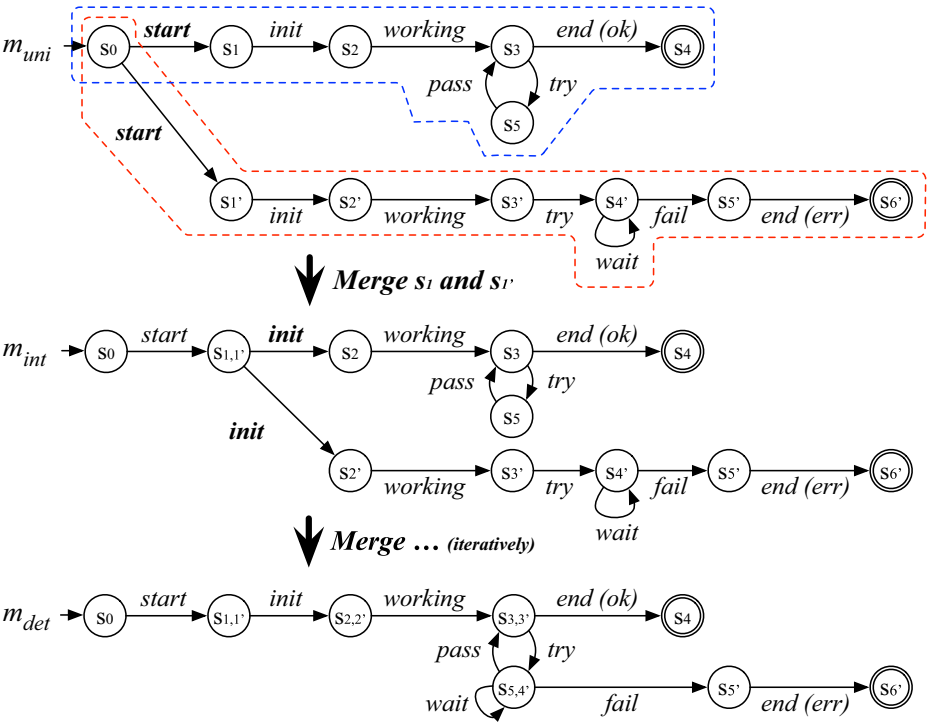

Fig. 6 Illustration of the determinization using state merges for the system model build for the running example logs. The labels of nondeterministic transitions are highlighted in bold

transformed through the last stage of *PRINS*, i.e., determinization, described in detail in Section 4.4.

4.4 Determinization

The last stage of *PRINS* post-processes the model yielded by the stitching stage for mainly converting a non-deterministic model into a deterministic one.

Through the projection, inference, and stitching stages, we already get a system model as an intermediate output. The non-determinism nature of such a model does not represent an issue in many use cases (e.g., program comprehension (Cook and Wolf 1998), test case generation Fraser and Walkinshaw, 2012). However, especially when a model is used as an "acceptor" of (the behavior recorded in) a log (e.g., in the case of anomaly detection Chandola et al., 2009), determinism is important for efficient checking. To broaden the use cases of *PRINS*, we propose this determinization stage as an optional post-processing in *PRINS*.

The simplest way of converting a non-deterministic model into a deterministic one is using standard algorithms, such as the powerset construction (Hopcroft et al. 2006), that guarantee the equivalence between the non-deterministic model provided in input and the deterministic one returned as output. However, the worst-case complexity of the powerset construction is exponential in the size of the non-deterministic model, making it an impracticable solution for many applications.

To tackle this issue, we introduce a new approach inspired by the heuristic-based determinization approach proposed by Damas et al. (2005). Unlike the powerset construction, their heuristic-based approach recursively merges the target states of non-deterministic transitions starting from the given state of the input model. While the idea of this approach is intuitive, since it simply merges states during the process of determinization, it may *generalize* the model being determinized, meaning the determinized model may accept additional logs that are not accepted by the original non-deterministic model. Our preliminary evaluation found that this simple strategy of merging states can produce an *over*-generalized model by merging too many states, especially when there are already many non-deterministic transitions in the input model. To avoid such over-generalization, we propose a new algorithm, called *Hybrid Determinization with parameter u* (HD_u), by combining ideas from the heuristic-based determinization and the powerset construction methods.

Our HD_u merges the target states of non-deterministic transitions, similar to the heuristic-based determinization. However, to prevent over-generalization it applies a heuristic: HD_u does not merge a state with other states[3] if the former has already been merged u times. The rationale behind this heuristic is to prevent the merging of too many states, which causes the over-generalization. If non-deterministic transitions remain because their target states are restricted from being merged because of the value of u, HD_u uses the powerset construction to remove the remaining non-determinism while preserving the level of the generalization. The larger the u value is used, the more the model can be generalized. The u value can also be seen as the weight between the heuristic-based determinization and the powerset construction; HD_∞ is the same as the heuristic-based determinization, while HD_0 is the same as the powerset construction.

Algorithm 3 shows the pseudocode of HD_u. It takes as input a non-deterministic model m_n and a threshold u; it returns a deterministic model m_d that can accept all logs that can be accepted by m_n.

Algorithm 3 Hybrid Determinization (HD_u).

```
   Input  : Model m_n
            Threshold u
   Output: Model m_d
1  Model m_d ← copy(m_n)
2  Set of States S_n ← getTargetStatesWithLimit(m_d, u)
3  while S_n ≠ ∅ do
4  |  mergeStates(m_d, S_n)
5  |  S_n ← getTargetStatesWithLimit(m_d, u)
6  if isNonDeterministic(m_d) then
7  |  m_d ← standardDeterminize(m_d)
8  return m_d
```

The algorithm iteratively merges the set of states S_n in m_d as determined by ***getTargetStatesWithLimit*** (described below) until it is empty (lines 3–5). After the iteration ends, if m_d is still non-deterministic, the algorithm removes all the remaining non-determinism using the powerset construction (lines 6–7). The algorithm ends by returning m_d.

[3]Regardless of the number of states to be merged, multiple states can be merged into one state at once, not incrementally.

The heuristic to avoid the over-generalization is mainly implemented in function ***getTargetStatesWithLimit*** (whose pseudocode is shown in Algorithm 4). It takes as input a non-deterministic model m_n and a threshold u; it returns a set of states S_n to be merged to reduce non-determinism in m_n, which does not contain the states that are restricted from being merged because of the threshold u.

Algorithm 4 *getTargetStatesWithLimit*.

```
Input  : Model m_n
         Threshold u
Output: Set of States S_n
1 foreach Guarded Transition gt ∈ getAllGT(m_n) do
2     Set of States S_t ← getTargetStates(gt)
3     Set of States S_n ← removeAlreadyMergedStates(S_t, u)
4     if |S_n| > 1 then
5         return S_n
6 return ∅
```

For each guarded transition gt in the transition relation of m_n (lines 1–5), the algorithm gets the set of target states S_t (line 2), removes the states that have already been merged u times from S_t to build S_n (line 3), and returns S_n (and ends) if it has more than one state (lines 4–5). If there is no such S_n for all guarded transitions, the algorithm ends by returning $S_n = \emptyset$ (line 6).

For example, let us consider the case where the ***getTargetStatesWithLimit*** algorithm begins the iteration for a non-deterministic (guarded) transition whose target states are s_{abc}, s_d, and s_e, with $u = 1$. If s_{abc} was generated by merging three states s_a, s_b, and s_c, S_t becomes $\{s_{abc}, s_d, s_e\}$ but S_n becomes $\{s_d, s_e\}$ because $removeAlreadyMergedStates$ excludes s_{abc} (since it has been already merged once, given $u = 1$). Since $|S_n| = 2$, the algorithm ends by returning $S_n = \{s_d, s_e\}$.

Figure 6 shows how HD works for our running example. Recall that m_{uni} is the intermediate output of the stitching stage. Starting from the initial state, HD iteratively merges the target states of non-deterministic transitions, such as s_1 and s'_1 in m_{uni} and then s_2 and s'_2 in m_{int}, until no more non-deterministic transition remains; the resulting model is m_{det}. We can see that m_{det} is exactly the same as (i.e., is graph-isomorphic to) the ideal model m_S in Fig. 2.

Notice that HD_u causes a reduction in size of the models since it merges the target states of non-deterministic transitions in the course of its heuristic determinization. However, a more important question is to what extent the accuracy of the resulting models varies because of size reduction. In our empirical evaluation, we will assess the impact of using HD_u on the accuracy of the inferred models in *PRINS* with respect to the value of threshold u. We will also investigate the execution time of HD_u and devise practical guidelines for choosing the value of u (see Section 5).

5 Evaluation

In this section, we report on the evaluation of the performance of *PRINS* in generating models of a component-based system from system logs.

First, we assess the execution time of *PRINS* in inferring models from large execution logs. This is the primary dimension we focus on since we propose *PRINS* as a viable alternative to state-of-the-art techniques for processing large logs. Second, we analyze how accurate the models generated by *PRINS* are. This is an important aspect because it is orthogonal to scalability but has direct implications on the feasibility of using the generated models to support software engineering tasks (e.g., test case generation). However, the execution time of *PRINS* and the accuracy of the models generated by *PRINS* might depend on its configuration, i.e., the number of parallel inference tasks in the inference stage (see Section 4.2) and the parameter u of HD_u in the determinization stage (see Section 4.4). Therefore, it is important to investigate the best configurations of *PRINS* before comparing it to state-of-the-art techniques.

Summing up, we investigate the following research questions:

RQ1: *How does the execution time of PRINS change according to the parallel inference tasks in the inference stage?*
RQ2: *How does the execution time of HD_u change according to parameter u?*
RQ3: *How does the accuracy of the models (in the form of gFSMs) generated by HD_u change according to parameter u?*
RQ4: *How fast is PRINS when compared to state-of-the-art model inference techniques?*
RQ5: *How accurate are the models generated by PRINS when compared to those generated by state-of-the-art model inference techniques?*

5.1 Benchmark and Settings

To evaluate *PRINS*, we assembled a benchmark composed of logs extracted from two sources: the LogHub project (He et al. 2020) and a personal computer (PC) running desktop business applications on a daily basis. Table 1 lists the systems we included in the benchmark (grouped by source) and provides statistics about the corresponding logs: the number of components (column *# Cmps*), the number of logs[4] (column *# Logs*), the number of event templates (column *# Tpls*), and the total number of log entries[5] (column *# Entries*).

LogHub (He et al. 2020) is a data repository containing a large collection of structured logs (and the corresponding event templates) from 16 different systems. Among them, we selected the logs of the five systems based on two conditions: (1) the component (name or ID) for each log entry is available in the logs; (2) the number of logs for each system is more than 10.

We set condition #1 because *PRINS* targets component-based systems; as for condition #2, we require a minimum number of logs to validate the accuracy as part of RQ2 (see Section 5.6).

To increase the diversity of our benchmark logs, we also included the logs of a personal computer running daily for office use. We collected the logs through the built-in

[4]In LogHub (He et al. 2020), the original dataset for HDFS contains 575061 logs (distinguishable by block-ids), but we used only 1000 logs that are randomly sampled from all logs, since we found that the 1000 logs are representative enough, as they contain all event templates that appear in all logs.

[5]When additional logging level information (e.g., `info`, `warn`, `debug`, `error`) was available for each log entry, we only used the log entries of the two main levels, i.e., `info` and `error`, as the others provide unnecessary details (e.g., the status of a specific internal variable) for building behavioral models.

Table 1 Subject systems and logs

Source	System	# Cmps	# Logs	# Tpls	# Entries	Conf
LogHub (He et al. 2020)	Hadoop	19	68	41	3575	1.00
	HDFS	8	1000	16	18741	0:95
	Linux	31	42	115	11259	0:78
	Spark	11	217	21	67725	0:98
	Zookeeper	18	36	40	25298	0:91
PC	CoreSync	54	1418	204	30223	0:94
	NGLClient	27	42	70	892	0.89
	Oobelib	12	250	147	56557	0:89
	PDApp	10	787	75	47394	0:87

`Console.app` application of macOS 10.15. Among the many logs available on the PC, we selected those fulfilling the same two conditions stated above, ending up with four systems. Additionally, to identify the events templates of the unstructured log messages in these logs, we first used state-of-the-art tools for log message format identification (i.e., Drain (He et al. 2017) and MoLFI (Messaoudi et al. 2018)) to compute an initial set of templates and then manually refined them, e.g., by collapsing similar templates into a single one. All the structured logs (anonymized to hide sensitive information) are available online (see Section 5.8).

To additionally evaluate whether the benchmark logs are sufficient to infer models that faithfully represent actual system behaviors, following another state-of-the-art model inference study (Emam and Miller 2018), we computed log confidence scores using the formula provided by Cohen and Maoz (2015). Briefly speaking, a low confidence score (e.g., ≤ 0.2) indicates that the logs are not sufficient, and therefore the model inferred from the logs are likely not to be compatible with the actual behaviors of the system under analysis. On contrary, a high confidence score (e.g., ≥ 0.85) indicates that the logs are probably sufficient for the inferred model to faithfully represent the actual system behaviors. Column *Conf* in Table 1 shows the confidence scores calculated for our benchmark logs. The values suggest that the logs are mostly sufficient to infer faithful models. Although the confidence score for Linux (0.78) is lower than the other benchmarks scores, the Linux logs are from an existing benchmark (He et al. 2020) and cannot therefore be improved. Furthermore, since our main focus is to compare *PRINS* and other model inference techniques using the same logs, having a somewhat moderate confidence score is not a major threat to the validity of our experiments.

We conducted our evaluation on a high-performance computing platform[6], using nodes equipped with Dell C6320 units (2 Xeon E5-2680v4@2.4 GHz, 128 GB). We allocated four cores and 16 GB per job.

[6]The experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg (Varrette et al. 2014) (see https://hpc.uni.lu for more details).

5.2 RQ1: Parallel Inference

5.2.1 Methodology

To answer RQ1, we assessed the execution time of *PRINS* with different parallelization configurations for its inference stage. Specifically, we varied the maximum number of parallel workers (i.e., the maximum number of parallel inference tasks) from one to four in steps of one to investigate the relationship between the maximum number of parallel workers and the execution time of *PRINS*. For example, when the number is set to four, at most four workers are running in parallel to infer four component models at the same time in the inference stage of *PRINS*.

To infer individual component models in the inference stage of *PRINS*, we used MINT (Walkinshaw et al. 2016), a state-of-the-art model inference tool. We selected MINT because other tools are either not publicly available or require additional information other than just logs (e.g., source code or architectural design documents). In all experiments, we used the same configuration of MINT (i.e., minimum state merge score $k = 2$ and AdaBoost as data classifier algorithm), which we set based on the one used in a previous study (Walkinshaw et al. 2016) conducted by the authors of MINT.

For each system in our benchmark, we ran the four configurations of *PRINS* to infer a system model from the same logs and measured their execution time. To account for the randomness in measuring execution time, we repeated the experiment 10 times.

We remark that we disabled the determinization stage of *PRINS* because it is not the main focus of RQ1. Determinization configurations will be comprehensively investigated in RQ2 and RQ3.

5.2.2 Results

Figure 7 shows the relationship between the maximum number of parallel inference tasks (workers) and the execution time of *PRINS*. None of the configurations was able to infer a

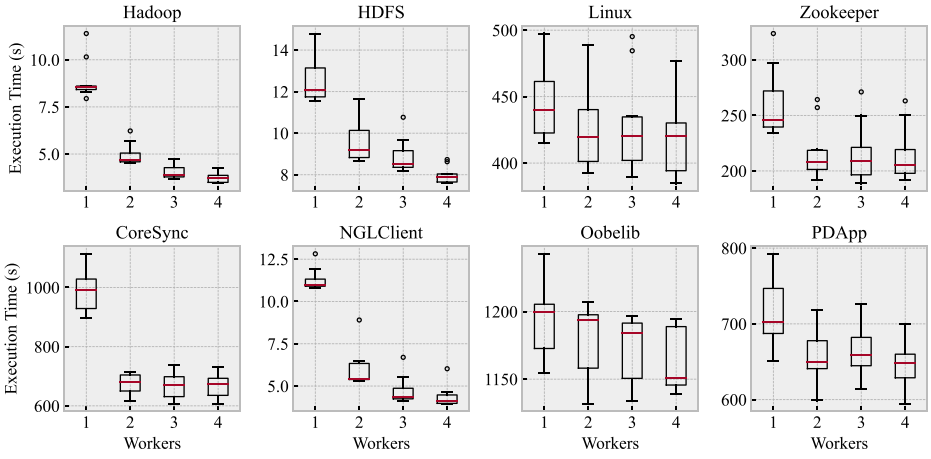

Fig. 7 Relationship between the maximum number of parallel workers and the execution time of *PRINS*

model for Spark on all ten executions due to out-of-memory errors. This occurred because MINT (used by *PRINS* for inferring individual component models) could not process the (huge) log of a component that is responsible for producing about 97% of all log messages of the system.

For all systems in our benchmark, it is clear that execution time decreases as the maximum number of parallel inferences increases. This is consistent with the general expectation for parallelization.

However, doubling the maximum number of parallel inferences does not decrease the execution time in half. For example, there is no clear difference in execution time between two workers and four workers for Linux, Zookeeper, CoreSync, and PDApp. A detailed analysis of the results found that it is mainly because there are at most two major components that take up more than 70% of all log messages. For example, Linux has two major components that represent around 50% and 34% of all log messages, while the third-largest component takes up only 9.4% of all messages. This implies that, for systems like Linux, inferring component models is fast enough, except for a few major components, and therefore having more than three parallel workers does not significantly reduce execution time.

The answer to RQ1 is that the execution time of *PRINS* can be significantly reduced by the parallel inference of individual component models. However, the magnitude of the reduction in execution time is not linear with respect to the maximum number of parallel inferences, because not all components are equally sized in their logs. In practice, an engineer can set the maximum number of parallel inferences considering both available resources (e.g., the number of CPUs and the total size of memory) and the log size distribution of components.

5.3 RQ2: Execution Time of Hybrid Determinization

5.3.1 Methodology

To answer RQ2, we assessed the execution time of HD_u with different parameter values for u. Specifically, we varied the value of u from one to ten in steps of one to investigate the relationship between the value of u and the execution time of HD_u. To additionally compare HD_u to the standard powerset construction, we also set $u = 0$ (see Section 4.4 for more details).

For each system in our benchmark, we first ran *PRINS* without the determinization stage to infer a non-deterministic system model. *PRINS* internally used the same configuration of MINT as used in RQ1. For each non-deterministic model, we ran HD_u for all $u = 0, 1, \ldots, 10$ and measured their execution time. To account for the randomness in measuring execution time, we repeated the experiment 10 times.

5.3.2 Results

Figure 8 shows the relationship between the value of u and the execution time of HD_u. We have no results for Spark because its non-deterministic system model was not available for the reasons explained in Section 5.2.2.

For all the cases in which *PRINS* completed their execution for generating non-deterministic system models, HD_u (with $u \geq 1$) took less than a minute. This implies that our hybrid determinization can efficiently determinize non-deterministic models generated by *PRINS*. On the other hand, the powerset construction (i.e., $u = 0$) took more

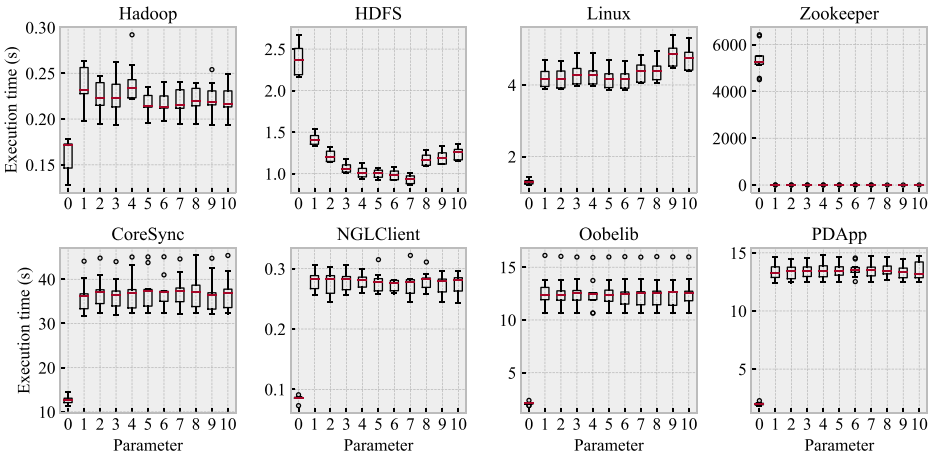

Fig. 8 Relationship between the value of u and the execution time of HD_u

than an hour for Zookeeper. This is due to the worst-case complexity of the powerset construction as discussed in Section 4.4. Interestingly, for the same non-deterministic model of Zookeeper, using $u \geq 1$ significantly reduces the determinization time. This clearly highlights the benefit of hybrid determinization, i.e., combining the powerset construction and the heuristic-based determinization.

The answer to RQ2 is that the execution time of HD_u is practically the same for all $u \geq 1$ because all of them are completed in less than a minute. This implies that the best value of u can be selected mainly based on the accuracy of the resulting models, which will be investigated in RQ3. On the other hand, the powerset construction is indeed very time-consuming in extreme cases, which is consistent with its well-known theoretical worst-case complexity. Since this cannot be predicted before running the determinization algorithms, in practice, we can conclude that HD_u with $u \geq 1$ is to be recommended over the standard powerset construction.

5.4 RQ3: Accuracy of Models Generated by Hybrid Determinization

5.4.1 Methodology

To answer RQ3, we first ran *PRINS* without the determinization stage to infer a non-deterministic system model for each system in our benchmark, as we did for RQ2. For each of the non-deterministic models, we then ran HD_u for all $u = 0, 1, \ldots, 10$ and measured the accuracy of the deterministic models generated by HD_u.

We measured the accuracy in terms of *recall*, *specificity*, and Balanced Accuracy (BA), following previous studies (Damas et al. 2005; Walkinshaw et al. 2016; Mariani et al. 2017; Emam and Miller 2018) in the area of model inference. Recall measures the ability of the inferred models of a system to accept "positive" logs, i.e., logs containing feasible behaviors that the system may exhibit. Specificity measures the ability of the inferred models to reject "negative" logs, i.e., logs containing behaviors that the system cannot exhibit. BA measures the balance between recall and specificity and provides the summary of the two.

However, it is intrinsically difficult to evaluate the accuracy of inferred models when there is no ground truth, i.e., reference models. To address this issue, we computed the

metrics by using the well-known k-fold cross validation (CV) method with $k = 10$, which has also been used in previous model inference studies (Walkinshaw et al. 2016; Mariani et al. 2017; Emam and Miller 2018). This method randomly partitions a set of logs into k non-overlapping folds: $k-1$ folds are used as "training set" from which the model inference tool infers a model, while the remaining fold is used as "test set" to check whether the model inferred by the tool accepts the logs in the fold. The procedure is repeated k times until all folds have been considered exactly once as the test set. For each fold, if the inferred model successfully accepts a positive log in the test set, the positive log is classified as True Positive (TP); otherwise, the positive log is classified as False Negative (FN). Similarly, if an inferred model successfully rejects a negative log in the test set, the negative log is classified as True Negative (TN); otherwise, the negative log is classified as False Positive (FP). Based on the classification results, we calculated $recall = \frac{|TP|}{|TP|+|FN|}$, $specificity = \frac{|TN|}{|TN|+|FP|}$, and the BA as the average of the recall and the specificity.

As done in previous work (Walkinshaw et al. 2016; Mariani et al. 2017; Emam and Miller 2018), we synthesized negative logs from positive logs by introducing small changes (mutations): (1) swapping two randomly selected log entries, (2) deleting a randomly selected log entry, and (3) adding a log entry randomly selected from other executions. The changes should be small, because the larger the change is, the easier an inferred model can detect the deviation of negative logs. To further increase the probability[7] that a log resulting from a mutation contains invalid behaviors of the system, we checked whether the sequence of entries around the mutation location (i.e., the mutated entries and the entries immediately before and after the mutants) did not also appear in the positive logs.

5.4.2 Results

Figure 9 shows the relationship between the value of u and the accuracy of the deterministic models generated by HD_u. Again, Spark is not shown for the reasons explained in Section 5.2.2.

For Hadoop, CoreSync, NGLClient, Oobelib, and PDApp, there is no change in recall, specificity, and BA when the value of u changes. This means that, for these five systems, the accuracy of deterministic models generated by HD_u does not change when the value of parameter u changes (for $u = 0, 1, \ldots, 10$). Furthermore, considering the fact that the powerset construction (i.e., HD_0) guarantees the equivalence between the non-deterministic model provided in input and the deterministic one returned as output, identical accuracy for $u = 0, 1, \ldots, 10$ also implies that, regardless of the value of u, HD_u can convert non-deterministic models into deterministic ones without sacrificing model accuracy for five out of the eight systems in our benchmark, regardless of the value of u.

For HDFS, Linux, and Zookeeper, as the value of u increases, recall values increase while specificity values decrease. This means that, for the deterministic models generated by HD_u, if we increase the value of u, then the ability to correctly accept positive logs is improved whereas the ability to correctly reject negative logs is diminished. This is intuitive because increasing the value of u merges more states to remove non-deterministic transitions, yielding a generalized model that accepts more logs than the original, non-deterministic model.

[7]Recall that there are no reference models for the subject systems, and therefore we cannot verify if a synthesized log correctly contains an invalid system behavior.

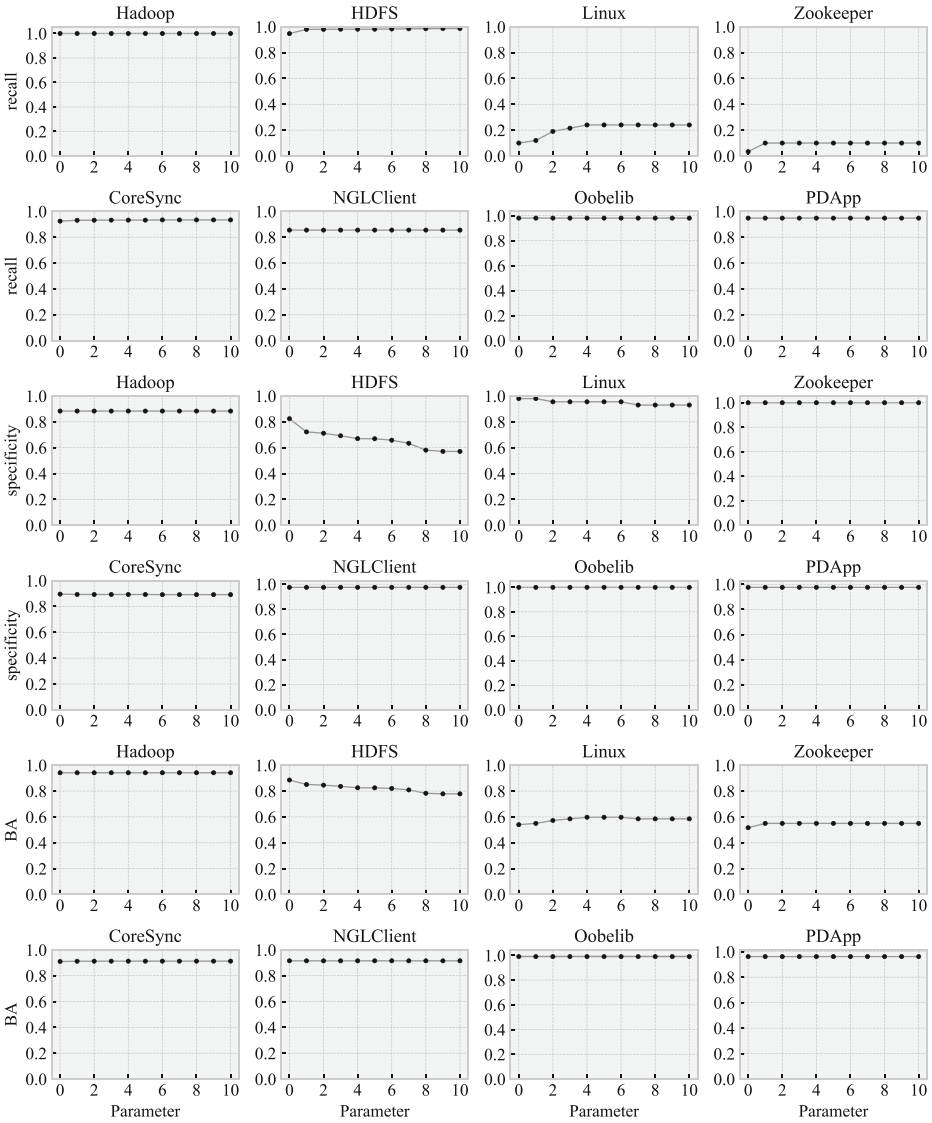

Fig. 9 Relationship between the value of u and the accuracy of the models generated by HD_u

However, we can distinguish the increase in recall and the decrease in specificity, because the former happens when the recall values of the input non-deterministic models are close to zero (i.e., Linux and Zookeeper), whereas the latter happens when the specificity values of the non-deterministic models are around 0.8. Since the non-deterministic models of Linux and Zookeeper were already incapable of correctly accepting positive logs, slightly improving them with determinization is not practically significant. In fact, the logs of Linux and Zookeeper were already inadequate for model inference in general, which will be discussed in detail in Section 5.6.2. On the other hand, the decrease in specificity for HDFS is

significant for HD_u since it should preserve the high specificity of the non-deterministic model provided in input as much as possible. As a result, practically speaking, the smaller the value of u, the better. Indeed, this supports our idea that using u to limit over-generalization in hybrid determinization is helpful to avoid a significant accuracy loss.

The answer to RQ3 is that, for five out of eight systems in our benchmark, the value of u does not affect the accuracy of the deterministic models generated by HD_u. However, for one system, the accuracy practically decreases as the value of n increases. Additionally considering the high execution time of HD_0 (RQ2), we can therefore conclude that $u = 1$ is the best configuration trade-off for HD_u in terms of both execution time and accuracy in practice.

5.5 RQ4: Execution Time of *PRINS* Compared to State-of-the-Art

5.5.1 Methodology

To answer RQ4, we assessed the scalability of *PRINS*, in terms of execution time, in comparison with MINT (Walkinshaw et al. 2016), the same tool that is used internally by *PRINS* to generate component-level models. In other words, we used *two* instances of MINT: the one used for the comparison in inferring system models; the other one used internally by *PRINS*. By doing this, we investigated to what extent the execution time of model inference can be improved by using the divide-and-conquer approach of *PRINS* compared to using a vanilla model inference.

Recall that *PRINS* can naturally infer many component models in parallel at the inference stage, which can further improve the execution time of *PRINS* as shown by the result of RQ1. To further investigate this aspect, we used *two* configurations of *PRINS*: *PRINS-P* where the parallel inference is enabled and *PRINS-N* where no parallelization is used. For *PRINS-P*, we set the maximum number of parallel inferences to four, based on the result of RQ1 and the number of allocated nodes (as described in Section 5.1). For both *PRINS-P* and *PRINS-N*, we used the determinization stage (i.e., HD_u), since MINT produces a deterministic model. For the value of u, we used $u = 1$ based on the results of RQ2 and RQ3.

We also varied the size of input logs to better understand the impact of using larger logs on the execution time of *PRINS* and MINT. To systematically increase such size while preserving the system behaviors recorded in individual logs, we duplicated each of the logs following the experiment design of Busany and Maoz (2016). For example, when the duplication factor is set to eight for the 250 logs (56 557 log entries) of Oobelib, each of the 250 logs is duplicated eight times, and therefore a total of $250 \times 8 = 2000$ logs ($8 \times 56\,557 = 452\,456$ log entries) are given as input both to *PRINS* and to MINT. Notice that the system characteristics, such as the number of components and the number of event templates, remain the same when using duplicated logs. Since MINT could not infer models for large logs due to out-of-memory failures or timeout (after 10 hours) in our preliminary evaluation, we only varied the duplication factor from 1 to 8 in steps of 1.

For each set of duplicated logs for each system in our benchmark, we ran MINT, *PRINS-P*, and *PRINS-N* to infer a deterministic system model from the same logs and measured the execution time of the tools. To account for the randomness in measuring execution time, we repeated the experiment three times and computed the average results.

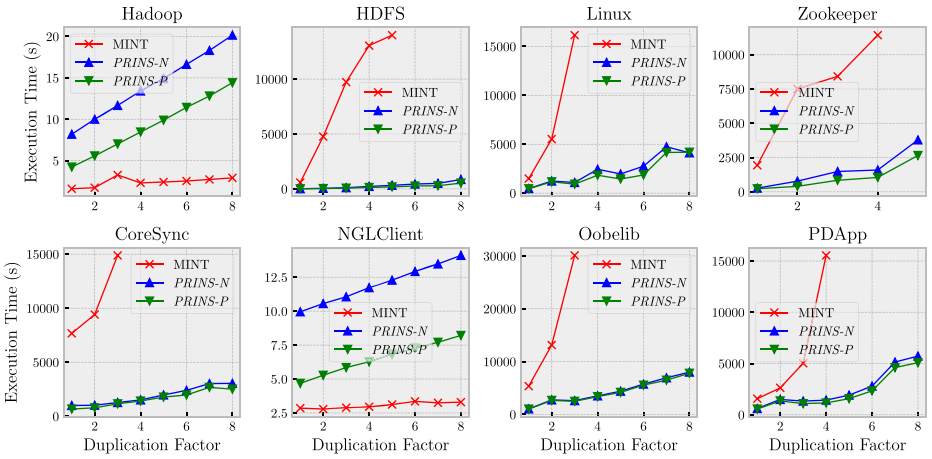

Fig. 10 Comparison between MINT, *PRINS-P*, and *PRINS-N* in terms of execution time for various log sizes (obtained by varying the duplication factor of the benchmark logs)

5.5.2 Results

Figure 10 shows the comparison results between MINT, *PRINS-P*, and *PRINS-N* in terms of execution time. Because MINT (both the standalone instance and the one used by *PRINS*) could not process the log of Spark, as explained in Section 5.2.2, we have no results for it. Also, due to the same reason, we have no results for Zookeeper with a duplication factor above 5.

For all the cases in which at least one of the tools completed their execution, we can see two distinct patterns in MINT's execution time: for all of the duplicated logs, MINT completed its execution only in two cases (Hadoop and NGLClient) out of eight; otherwise, MINT could not complete its execution (HDFS, Linux, Zookeeper, CoreSync, Oobelib, and PDApp).

However, for Hadoop and NGLClient, for which MINT completed its execution for all of the duplicated logs, we can see that MINT was quite fast (the execution time is less than 5s); this can be attributed to the small size of the logs (28 600 entries for Hadoop and 7136 entries for NGLClient even with a duplication factor of 8). When the (standalone) MINT instance is already fast, *PRINS* is actually slower than MINT due to the overhead for projection, stitching, and determinization. Nevertheless, even in these cases, using *PRINS* instead of MINT is still practical since *PRINS-P* took less than 15s. Also, we want to remark that such small logs are not really representative of the large logs targeted by *PRINS*.

For the remaining six cases (HDFS, Linux, Zookeeper, CoreSync, Oobelib, and PDApp) with larger log sizes, the execution time of MINT increases steeply as the duplication factor increases. Furthermore, with a duplication factor above a certain value (5 for HDFS, 4 for Zookeeper and PDApp, and 3 for Linux, CoreSync, and Oobelib), MINT could not complete its execution due to out-of-memory failures (Zookeeper) or timeouts after 10 hours (HDFS, Linux, CoreSync, Oobelib, and PDApp). In contrast, for the same logs, the execution time of *PRINS-N* increases slowly as the duplication factor increases, and there is no

case where *PRINS-N* could not complete its execution (except for Zookeeper in which the MINT instance internally used by *PRINS* for component model inference caused an out-of-memory failure when the duplication factor is greater than 5). This means that the scalability of model inference can be greatly improved by using the divide-and-conquer approach of *PRINS*. Furthermore, for Zookeeper with a duplication factor of 5, though MINT could not complete its execution due to out-of-memory failures, *PRINS*, on the other hand, completed successfully due to its divide-and-conquer approach.

Interestingly, for the six cases with larger log sizes, the difference between *PRINS-P* and *PRINS-N* is very small compared to the difference between *PRINS-N* and MINT. This means that the key factor in the scalability improvement of *PRINS* is the divide-and-conquer approach, not the parallel inference of component models.

The answer to RQ4 is that the divide-and-conquer approach of *PRINS* greatly improves the scalability of model inference for component-based system logs and can even enable model inference when MINT leads to out-of-memory failures.

5.6 RQ5: Accuracy of *PRINS* Compared to State-of-the-Art

5.6.1 Methodology

To answer RQ2, we assessed the accuracy of the models inferred both by *PRINS* and by MINT for each system in our benchmark, using the same configuration for *PRINS* and MINT used as part of RQ4. We measured the accuracy in terms of recall, specificity and BA as we did for RQ3 (see Section 5.4.1 for more details).

5.6.2 Results

The accuracy scores of *PRINS* and MINT are shown in Table 2. Under the *Recall* column, sub-columns *M* and *P* indicate the recall of MINT and *PRINS*, respectively, and sub-column Δ_R indicates the difference in recall between *PRINS* and MINT in percentage points (pp). The sub-columns under the *Specificity* and *Balanced Accuracy* columns follow the same

Table 2 Comparison between MINT (M) and $PRINS$ (P) in terms of accuracy

		Recall			Specificity			Balanced accuracy		
System	M	P	ΔR	M	P	ΔS	M	P	ΔB LDS	
Hadoop	1.00	1.00	0.0	0.91	0.88	−3.1	0.96	0.94	−1.5	0.015
HDFS	0.98	0.98	−0.1	0.37	0.72	34.9	0.68	0.85	17.4	0.007
Linux	0.36	0.12	−23.5	0.89	0.98	9.5	0.62	0.55	−7.0	0.561
Zookeeper	0.22	0.10	−11.7	0.93	1.00	7.5	0.57	0.55	−2.1	0.571
CoreSync	0.95	0.93	−1.6	0.85	0.89	4.2	0.90	0.91	1.3	0.048
NGLClient	0.86	0.86	0.0	1.00	0.98	−2.5	0.93	0.92	−1.3	0.195
Oobelib	0.98	0.98	0.0	1.00	1.00	0.0	0.99	0.99	0.0	0.016
PDApp	0.97	0.95	−2.3	0.98	0.98	−0.1	0.97	0.96	−1.2	0.014
Average	0.79	0.74	−4.9	0.87	0.93	6.3	0.83	0.83	0.7	0.178

Differences between recall (Δ_R), specificity (Δ_S), and balanced accuracy (Δ_B) values are expressed in percentage points (pp); $\mathcal{LDS}$ is the log-component diversity score

structure, with sub-column Δ_S indicating the difference in specificity between *PRINS* and MINT, and sub-column Δ_B indicating the difference in BA between *PRINS* and MINT. Again, none of the tools was able to infer a model for Spark, for the reasons explained in Section 5.5.2.

For all the cases in which the 10-fold CV completed without error, the average difference in BA between *PRINS* and MINT is only 0.7 pp, meaning that, on average, *PRINS* is as accurate as MINT in inferring system models in terms of BA. However, the average difference in recall between *PRINS* and MINT is -4.9 pp, while the average difference in specificity between *PRINS* and MINT is 6.3 pp. This implies that, on average, *PRINS* tends to infer models that are relatively less capable of accepting positive logs but more capable of rejecting negative logs than those inferred by MINT. The intuitive explanation is that a model built by *PRINS* could be, in certain cases discussed below, more specific to the flows of events recorded in individual input logs, due to the way *PRINS* builds the model. As described in Section 4.3, *PRINS* first builds an intermediate system-level model *for each execution log* and then merges these intermediate models by merging only their initial states at the end of the stitching. Though determinization after stitching might further merge the other states for removing non-determinism, it does so only for the states related to non-deterministic transitions. Therefore, the execution-specific flows of events captured in the intermediate system-level models can be maintained (without being merged with the others) in the final system model built by *PRINS*. In contrast, since MINT infers a model *for all system execution logs at once*, it tends to merge the execution-specific flows of events to a larger extent than *PRINS*. As expected, such characteristics also impact the size of inferred models. As shown in Table 3, the models inferred by *PRINS* have on average 3.6 times more states and 5.5 times more transitions than the models inferred by MINT. Since larger models are more difficult to manually analyze and comprehend, this might be interpreted as a drawback of *PRINS*. However, the models inferred by MINT are already too large to be manually analyzed and understood, especially for systems with large logs. Thus, automated techniques, such as model abstraction (Polyvyanyy et al. 2008), should be utilized in practice anyway. Furthermore, inferred models can be used for other important applications, such as test case generation (Fraser and Walkinshaw 2012) and anomaly detection (Chandola et al. 2009), which do not require minimally sized models. Therefore, the increased model size can be considered acceptable given the significant scalability improvement reported in Section 5.5.

Table 3 Comparison between MINT and *PRINS* in terms of model size

	States			Transitions		
System	MINT	*PRINS*	ratio	MINT	PRINS	ratio
Hadoop	67	65	1.0	70	66	0.9
HDFS	76	392	5.2	177	1308	7.4
Linux	342	1990	5.8	476	3322	7.0
Zookeeper	376	3184	8.5	553	10667	19.3
CoreSync	3876	7524	1.9	4318	10798	2.5
NGLClient	148	154	1.0	160	195	1.2
Oobelib	447	1195	2.7	545	1484	2.7
PDApp	1301	3523	2.7	1466	3801	2.6
Average	829	2253	3.6	971	3955	5.5

Looking at the results for individual systems, results differ significantly in terms of Δ_R and Δ_S and it is important to understand why to draw conclusions. For instance, for HDFS, the value of Δ_S is high (34.9pp), while the value of Δ_R is negligible. This shows that *PRINS*, compared to MINT, can significantly increase the accuracy of the inferred models by increasing their ability to correctly reject negative logs, without compromising their ability to correctly accept positive logs.

On the other hand, for Linux and Zookeeper, the values of Δ_R are negative and practically significant (-23.5pp for Linux and -11.7pp for Zookeeper) while the values of Δ_S are positive and practically significant as well (34.9pp for Linux and 7.5pp for Zookeeper). Furthermore, the recall values of both MINT and *PRINS* are relatively lower for Linux and Zookeeper compared to the recall values for the other systems. In terms of the 10-fold CV, this means that the positive logs in the test set are not properly accepted by the models inferred from the logs in the training set for Linux and Zookeeper. Experimentally, this is mainly due to the logs in the training set being *too different* from the logs in the test set, this being caused by the highly diverse logs of Linux and Zookeeper overall. From a practical standpoint, this implies that, regardless of the model inference technique, a model inferred from existing logs may not be able to correctly accept unseen (but positive) logs if the latter are too different from the former. However, for the reasons mentioned above, the issue of highly diverse logs has a moderately larger impact on *PRINS* than on MINT. Practical implications are discussed below.

Before running model inference, to effectively predict and avoid cases where *PRINS* is likely to be worse than MINT and where both techniques fare poorly, we propose a new and practical metric to measure the diversity of logs. Our log diversity metric is based on the combination of components appearing in the individual logs because (1) *PRINS* targets component-based systems considering not only the individual components' behaviors but also their interactions, (2) it is much simpler than using, for example, the flows of log entries in the logs, and (3) it does not require any extra information other than the logs. More formally, let L be a set of logs of a system and let $C(l)$ be the set of components appearing in a log $l \in L$. We define *log-component diversity score* ($\mathcal{LDS}$) of the system logs L_{sys} as $\mathcal{LDS}(L_{sys}) = \frac{U-1}{N-1}$, where $U = |\{C(l) \mid l \in L_{sys}\}|$ (i.e., the total number of unique $C(l)$s for all $l \in L_{sys}$) and $N = |L_{sys}|$ (i.e., the total number of logs in L_{sys}). In other words, $\mathcal{LDS}$ indicates the ratio of logs that are unique (i.e., different from the others) in terms of the set of components appearing in the individual logs, ranging between 0 and 1; the higher its value, the higher the diversity of the logs in terms of recording different component interactions. For instance, $\mathcal{LDS}(L_S) = 0$ for our running example logs $L_S = \{l_1, l_2\}$ because $N = 2$ and $U = |\{C(l_1), C(l_2)\}| = 1$ (since $C(l_1) = C(l_2) = \{\texttt{Master}, \texttt{Job}\}$). This means that L_S is not diverse at all in terms of the appearing components. Notice that $\mathcal{LDS}$ is a characteristic of logs, which can be calculated before model inference takes place.

We measured $\mathcal{LDS}$ for the logs of each system in our benchmark. Column $\mathcal{LDS}$ in Table 2 shows the results. We can see that the resulting $\mathcal{LDS}$ values of Linux (0.561) and Zookeeper (0.571) are much higher than those of the other systems, which range between 0.007 (HDFS) and 0.195 (NGLClient). This confirms that $\mathcal{LDS}$ can be effectively used to predict whether the models inferred from the existing logs can correctly accept unseen (but positive) logs or not before running model inference.

In practice, if $\mathcal{LDS}$ is high (e.g., > 0.2) for the logs of a system, it implies that these logs do not sufficiently exercise, in a comprehensive way, the potential behaviors of the system. As a result, there is a high probability that many component interactions have not been recorded or too rarely so. Therefore, an engineer can address this problem by collecting more system logs until $\mathcal{LDS}$ is low enough.

The answer to RQ5 is that, compared to MINT, *PRINS* tends to infer models that are more capable of rejecting negative logs (i.e., yielding a higher specificity value) while sometimes being less capable of accepting positive logs (i.e., yielding a lower recall value). The latter happen anyway only in cases where logs are not adequate for both techniques to work well. In practice, an engineer can compute the diversity score of the logs before running model inference, and easily determine whether more logs should be collected, either through testing or usage, until the score is acceptable.

5.7 Discussion and Threats to Validity

From the results above, we conclude that *PRINS* is an order of magnitude faster than MINT in model inference for component-based systems, especially when the input system logs are large and the individual component-level logs are considerably smaller than the system logs, without significantly compromising the accuracy of the models. Furthermore, since the large majority of modern software systems is composed of many "components", which can be modules, classes, or services, depending on the context, the logs typically encountered in practice will satisfy the best conditions for *PRINS* to fare optimally: the system logs are large but the individual component-level logs are considerably smaller. There are situations where *PRINS* exhibits a poorer recall than MINT. However, this is the case when the system logs are inadequate for model inference in general, regardless of the technique, and we have proposed a way to detect such situations and remedy the problem.

One drawback of the divide-and-conquer approach in *PRINS* is the increased size of inferred models. In this sense, *PRINS* can be seen as sacrificing model size for improving the execution time of model inference. Nevertheless, it is worth to note that *PRINS* does not significantly compromise the accuracy of the inferred models. Furthermore, given the significant execution time reduction in model inference on large logs, increasing model size can be considered acceptable.

In terms of threats to validity, using a specific model inference tool (MINT) is a potential factor that may affect our results. However, we expect that applying other model inference techniques would not change the trends in results since the fundamental principles underlying the different model inference techniques are very similar. Furthermore, MINT is considered state-of-the-art among available tools. Nevertheless, an experimental comparison across alternative tools would be useful and is left for future work.

We used k-fold cross validation to evaluate the accuracy of inferred models due to the lack of ground truth (i.e., reference models) for our benchmark systems. Therefore, the computed accuracy scores might not faithfully represent the similarity between the inferred models and their (unknown) ground truths, especially when the collected logs do not sufficiently represent the system behaviors. To mitigate this issue, we calculated the log-confidence values, following existing studies, and these results suggested that the logs in our benchmarks are sufficient to derive faithfully inferred models. Furthermore, since the same logs are used for both *PRINS* and MINT, not relying on ground truth does not severely affect our empirical evaluation results.

5.8 Data Availability

The implementation of *PRINS* is available as a Python program. The replication package, including the benchmark logs and our implementation of *PRINS*, is at https://github.com/SNTSVV/PRINS.

6 Related Work

Starting from the seminal work of Biermann and Feldman (1972) on the *k-Tail* algorithm, which is based on the concept of state merging, several approaches have been proposed to infer a Finite State Machine (FSM) from execution traces or logs. *Synoptic* (Beschastnikh et al. 2011) uses temporal invariants, mined from execution traces, to steer the FSM inference process to find models that satisfy such invariants; the space of the possible models is then explored using a combination of model refinement and coarsening. *InvariMINT* (Beschastnikh et al. 2015) is an approach enabling the declarative specification of model inference algorithms in terms of the types of properties that will be enforced in the inferred model; the empirical results show that the declarative approach outperforms procedural implementations of *k-Tail* and *Synoptic*. Nevertheless, this approach requires prior knowledge of the properties that should hold on the inferred model; such a pre-condition cannot be satisfied in contexts (like the one in which this work is set) where the knowledge about the system is limited and the only information about the system is provided by logs. *mk-Tails* (Busany et al. 2019) is a generalization of the *k-Tail* algorithm from single to many parameters, which enables fine-grained control over the abstraction (generalization) on different subsets of the events. It allows users to deal with the trade-off between size and accuracy in model inference.

Other approaches infer other types of behavioral models that are richer than an FSM. *GK-tail+* (Mariani et al. 2017) infers guarded FSM (gFSM) by extending the *k-Tail* algorithm and combining it with Daikon (Ernst et al. 2007) to synthesize constraints on parameter values; such constraints are represented as guards of the transitions of the inferred model. *MINT* (Walkinshaw et al. 2016) also infers a gFSM by combining EDSM (Evidence-Driven State Merging) (Cheng and Krishnakumar 1993) and data classifier inference (Witten et al. 2016). EDSM, based on the Blue-Fringe algorithm (Lang et al. 1998), is a popular and accurate model inference technique, which won the Abbadingo (Lang et al. 1998) competition; it is also utilized in DFASAT (Heule and Verwer 2013) that won the StaMinA competition (Walkinshaw et al. 2013). Data-classifier inference identifies patterns or rules between data values of an event and its subsequent events. Using data classifiers, the data rules and their subsequent events are explicitly tied together. *ReHMM* (Reinforcement learning-based Hidden Markov Modeling) (Emam and Miller 2018) infers a gFSM extended with transition probabilities, by using a hybrid technique that combines stochastic modeling and reinforcement learning. ReHMM is built on top of MINT; differently from the latter, it uses a specific data classifier (Hidden Markov model) to deal with transition probabilities.

Model inference has also been proposed in the context of distributed and concurrent systems. *CSight* (Beschastnikh et al. 2014) infers a communicating FSM from logs of vector-timestamped concurrent executions, by mining temporal properties and refining the inferred model in a way similar to *Synoptic*. *MSGMiner* (Kumar et al. 2011) is a framework for mining graph-based models (called Message Sequence Graphs) of distributed systems; the nodes of this graph correspond to Message Sequence Chart, whereas the edges are determined using automata learning techniques. This work has been further extended (Kumar et al. 2012) to infer (symbolic) class level specifications. However, these approaches require the availability of channel definitions, i.e., which events are used to send and receive messages among components.

Liu and Dongen (Liu et al. 2016) use a *divide-and-conquer* strategy, similar to the one in our *PRINS* approach, to infer a system-level, hierarchical process model (in the form of a Petri net with nested transitions) from the logs of interleaved components, by leveraging the calling relation between the methods of different components. This approach assumes

the knowledge of the caller and callee of each component methods; in our case, we do not have this information and rely on the *leads-to* relation among log entries, computed from high-level architectural descriptions and information about the communication events.

Nevertheless, all the aforementioned approaches cannot avoid scalability issues due to the intrinsic computational complexity of inferring FSM-like models; the minimal consistent FSM inference from logs is NP-complete (Gold 1967) and all the more practical approaches are approximation algorithms with polynomial complexity.

One way to tackle the intrinsic scalability issue of (automata-based) model inference is to rely on distributed computing models, such as MapReduce (Dean and Ghemawat 2008), by transforming the sequential model inference algorithms into their corresponding distributed version. In the case of the *k-Tail* algorithm, the main idea (Wang et al. 2015) is to parallelize the algorithm by dividing the traces (sequences of log messages) into several groups, and then run an instance of the sequential algorithm on each of them. A more fine-grained version (Luo et al. 2017) parallelizes both the trace slicing and the model synthesis steps. Being based on MapReduce, both approaches require to encode the data to be exchanged between mappers and reducers in the form of key-value pairs. This encoding, especially in the trace slicing step, is application-specific; for instance, to correctly slice traces recorded by an online shopping system, different event parameter values, such as `user id`, `order id`, and `item id`, must be correctly identified and categorized from individual messages beforehand. Notice that this is more challenging than just identifying parameter values from free-formed messages, since different types of parameters must be distinguished. Hence, MapReduce cannot be used in contexts in which the system is treated as a black-box, with limited information about the data recorded in the log entries. Furthermore, though the approach can infer a FSM from large logs of over 100 million events, the distributed model synthesis can be significantly slower for $k \geq 3$ (of *k-Tail*), since the underlying algorithm is exponential in k.

Another way of taming scalability is to reduce the size of input logs by sampling them from the entire set of collected logs using statistical analysis and provide statistical guarantees on the inferred models. This is called *statistical log analysis* and was first presented by Busany and Maoz (2016). Its key idea is to iteratively sample new logs until the probability of adding new system behaviors into the model inferred by sampled logs is less than a given level of confidence threshold. While the idea of using statistical analysis to address the scalability of model inference is promising, as already noted by the authors, it is only applicable to *sequential* model inference algorithms, where each log can be processed independently (Busany and Maoz 2016). *PRINS*, on the other hand, is applicable to all model inference algorithms as only the inference target is changed from systems to components. Therefore, all model inference algorithms can benefit from using the divide-and-conquer approach in *PRINS*.

7 Conclusion

In this paper, we addressed the scalability problem of inferring the model of a component-based system from system logs, assuming that the only information available about the system is represented by the logs. Our approach, called *PRINS*, first infers a model of each system component from the corresponding logs; then, it merges the individual component models together taking into account the flow of events across components, as reflected in the logs. Our evaluation, performed on logs from nine datasets, has shown that *PRINS* can process large logs an order of magnitude faster than a publicly available and well-known

state-of-the-art technique without significantly compromising the accuracy of inferred models. While there are some cases where *PRINS* achieves a moderately lower recall than the state-of-the-art, this happens when the logs are inadequate for model inference in general, regardless of the technique. Furthermore, we have proposed an easy way to detect such cases and remedy the problem.

As part of future work, we plan to evaluate *PRINS* on different datasets, especially collected from real-world industrial applications, and to integrate it with other model inference techniques. We also aim to assess the effectiveness of the inferred models when applied to support software engineering activities, such as test case generation.

Declarations

Conflict of Interests The authors declare that they have no conflict of interest.

References

Aghajani E, Nagy C, Vega-Márquez OL, Linares-Vásquez M, Moreno L, Bavota G, Lanza M (2019) Software documentation issues unveiled. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE Press, Piscataway, pp 1199–1210

Beschastnikh I, Brun Y, Schneider S, Sloan M, Ernst MD (2011) Leveraging existing instrumentation to automatically infer invariant-constrained models. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011). ACM, New York, pp 267–277

Beschastnikh I, Brun Y, Ernst MD, Krishnamurthy A (2014) Inferring models of concurrent systems from logs of their behavior with CSight. In: Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, New York, pp 468–479

Beschastnikh I, Brun Y, Abrahamson J, Ernst MD, Krishnamurthy A (2015) Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. IEEE Trans Softw Eng 41(4):408–428

Biermann AW, Feldman JA (1972) On the synthesis of finite-state machines from samples of their behavior. IEEE Trans Comput C-21(6):592–597. https://doi.org/10.1109/TC.1972.5009015

Busany N, Maoz S (2016) Behavioral log analysis with statistical guarantees. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp 877–887. https://doi.org/10.1145/2884781.2884805

Busany N, Maoz S, Yulazari Y (2019) Size and accuracy in model inference. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE Press, Piscataway, pp 887–898. https://doi.org/10.1109/ASE.2019.00087

Chandola V, Banerjee A, Kumar V (2009) Anomaly detection: a survey. ACM Comput Surv 41(3). https://doi.org/10.1145/1541880.1541882

Cheng K, Krishnakumar AS (1993) Automatic functional test generation using the extended finite state machine model. In: Proceedings of the 30th Design Automation Conference (DAC 1993). ACM, New York, pp 86–91

Clarke Jr, EM, Grumberg O, Kroening D, Peled D, Veith H (2018) Model checking. MIT Press, Cambridge

Cohen H, Maoz S (2015) Have we seen enough traces? (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 93–103. https://doi.org/10.1109/ASE.2015.62

Cook JE, Wolf AL (1998) Discovering models of software processes from event-based data. ACM Trans Softw Eng Methodol 7(3):215–249. https://doi.org/10.1145/287000.287001
Damas C, Lambeau B, Dupont P, van Lamsweerde A (2005) Generating annotated behavior models from end-user scenarios. IEEE Trans Softw Eng 31(12):1056–1073. https://doi.org/10.1109/TSE.2005.138
Dean J, Ghemawat S (2008) Mapreduce: Simplified data processing on large clusters. Commun ACM 51(1):107–113
El-Masri D, Petrillo F, Guéhéneuc YG, Hamou-Lhadj A, Bouziane A (2020) A systematic literature review on automated log abstraction techniques. Inf Softw Technol 122:106276. https://doi.org/10.1016/j.infsof.2020.106276
Emam SS, Miller J (2018) Inferring extended probabilistic finite-state automaton models from software executions. ACM Trans Softw Eng Methodol 27(1). https://doi.org/10.1145/3196883
Ernst MD, Perkins JH, Guo PJ, McCamant S, Pacheco C, Tschantz MS, Xiao C (2007) The Daikon system for dynamic detection of likely invariants. Sci Comput Program 69(1):35–45
Fraser G, Walkinshaw N (2012) Behaviourally adequate software testing. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE Press, Piscataway, pp 300–309. https://doi.org/10.1109/ICST.2012.110
Gold EM (1967) Language identification in the limit. Inf Control 10(5):447–474
He P, Zhu J, Zheng Z, Lyu MR (2017) Drain: an online log parsing approach with fixed depth tree. In: 2017 IEEE International Conference on Web Services (ICWS). IEEE Press, Piscataway, pp 33–40. https://doi.org/10.1109/ICWS.2017.13
He S, Zhu J, He P, Lyu MR (2020) Loghub: A large collection of system log datasets towards automated log analytics. arXiv:2008.06448
Heule MJH, Verwer S (2013) Software model synthesis using satisfiability solvers. Empir Software Eng 18:825–856. https://doi.org/10.1007/s10664-012-9222-z
Hopcroft JE, Motwani R, Ullman JD (2006) Introduction to automata theory, languages and computation, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., USA
Kumar S, Khoo SC, Roychoudhury A, Lo D (2011) Mining message sequence graphs. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011). ACM, New York, pp 91–100
Kumar S, Khoo SC, Roychoudhury A, Lo D (2012) Inferring class level specifications for distributed systems. In: Proceedings of the 34th International Conference on Software Engineering (ICSE 2012). IEEE, Piscataway, pp 914–924
Lang KJ, Pearlmutter BA, Price RA (1998) Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In: Proceedings of the 4th International Colloquium on Grammatical Inference (ICGI 1998), LNCS, vol 1433. Springer, Berlin, pp 1–12
Liu C, van Dongen B, Assy N, van der Aalst WMP (2016) Component behavior discovery from software execution data. In: Proceedings of the Symposium Series on Computational Intelligence (SSCI 2016). IEEE, Piscataway, pp 1–8
Luo C, He F, Ghezzi C (2017) Inferring software behavioral models with mapreduce. Sci Comput Programm 145:13–36. https://doi.org/10.1016/j.scico.2017.04.004, http://www.sciencedirect.com/science/article/pii/S0167642317300795
Mariani L, Pezzè M, Santoro M (2017) Gk-tail+ an efficient approach to learn software models. IEEE Trans Softw Eng 43(8):715–738. https://doi.org/10.1109/TSE.2016.2623623
Messaoudi S, Panichella A, Bianculli D, Briand L, Sasnauskas R (2018) A search-based approach for accurate identification of log message formats. In: 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC). IEEE Press, Piscataway, pp 167–16710
Palmer JD, McAddis N (2019) Documentation as a cross-cutting concern of software. In: Proceedings of the 37th ACM International Conference on the Design of Communication, SIGDOC '19. Association for Computing Machinery, New York. https://doi.org/10.1145/3328020.3353949
Polyvyanyy A, Smirnov S, Weske M (2008) Process model abstraction: A slider approach. In: 2008 12th International IEEE Enterprise Distributed Object Computing Conference, pp 325–331. https://doi.org/10.1109/EDOC.2008.17
Rios N, Mendes L, Cerdeiral C, Magalhães APF, Perez B, Correal D, Astudillo H, Seaman C, Izurieta C, Santos G, Oliveira spínola R (2020) Hearing the voice of software practitioners on causes, effects, and practices to deal with documentation debt. In: Requirements engineering: Foundation for software quality. Springer International Publishing, Cham, pp 55–70
Varrette S, Bouvry P, Cartiaux H, Georgatos F (2014) Management of an academic hpc cluster: The ul experience. In: Proc. of the 2014 intl. Conf. on high performance computing & simulation (HPCS 2014). IEEE, Bologna, pp 959–967

Walkinshaw N (2018) mintframework. https://github.com/neilwalkinshaw/mintframework, accessed: 2020-03-05

Walkinshaw N, Bogdanov K, Damas C, Lambeau B, Dupont P (2010) A framework for the competitive evaluation of model inference techniques. In: Proceedings of the First International Workshop on Model Inference In Testing (MIIT 2010). ACM, New York, pp 1–9

Walkinshaw N, Lambeau B, Damas C, Bogdanov K, Dupont P (2013) Stamina: a competition to encourage the development and assessment of software model inference techniques. Empir Softw Eng 18(4):791–824

Walkinshaw N, Taylor R, Derrick J (2016) Inferring extended finite state machine models from software executions. Empir Softw Eng 21(3):811–853. https://doi.org/10.1007/s10664-015-9367-7

Wang S, Lo D, Jiang L, Maoz S, Budi A (2015) Scalable parallelization of specification mining using distributed computing. In: Bird C, Menzies T, Zimmermann T (eds) The Art and Science of Analyzing Software Data. Morgan Kaufmann, Boston, pp 623–648. https://doi.org/10.1016/B978-0-12-411519-4.00021-5, http://www.sciencedirect.com/science/article/pii/B9780124115194000215

Witten IH, Frank E, Hall MA, Pal CJ (2016) Data mining: Practical machine learning tools and techniques, 4th edn. Morgan Kaufmann, San Francisco

Zhu J, He S, Liu J, He P, Xie Q, Zheng Z, Lyu MR (2019) Tools and benchmarks for automated log parsing. In: Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, , ICSE-SEIP '10. IEEE Press, Piscataway, pp 121–130. https://doi.org/10.1109/ICSE-SEIP.2019.00021