Science of Computer Programming 145 (2017) 13–36

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Inferring software behavioral models with MapReduce

Chen Luo [a,b,1], Fei He [a,*], Carlo Ghezzi [c]

[a] Tsinghua National Laboratory for Information Science and Technology (TNList), Key Laboratory for Information System Security, Ministry of Education, School of Software, Tsinghua University, Beijing 100084, China
[b] University of California, Irvine, USA
[c] Politecnico di Milano, Italy

ARTICLE INFO

Article history:
Received 17 June 2016
Received in revised form 13 April 2017
Accepted 16 April 2017
Available online 24 April 2017

Keywords:
Model inference
Parametric trace
Log analysis
MapReduce

ABSTRACT

In the real world practice, software systems are often built without developing any explicit upfront model. This can cause serious problems that may hinder the almost inevitable future evolution, since at best the only documentation about the software is in the form of source code comments. To address this problem, research has been focusing on automatic inference of models by applying machine learning algorithms to execution logs. However, the logs generated by a real software system may be very large and the inference algorithm can exceed the processing capacity of a single computer.

This paper proposes a scalable, general approach to the inference of *behavior models* that can handle large execution logs via parallel and distributed algorithms implemented using the MapReduce programming model and executed on a cluster of interconnected execution nodes. The approach consists of two distributed phases that perform *trace slicing* and *model synthesis*. For each phase, a distributed algorithm using MapReduce is developed. With the parallel data processing capacity of MapReduce, the problem of inferring behavior models from large logs can be efficiently solved. The technique is implemented on top of Hadoop. Experiments on Amazon clusters show efficiency and scalability of our approach.

1. Introduction

Software behavior models play an important role in the whole life cycle of software systems. Through models, software engineers may gain a deep understanding of how a system behaves without dealing with the intricacies of the implementation. Although good software engineering practices suggest that models should be developed upfront, before deriving an implementation, reality shows that often models do not exist, or they are inconsistent with the implementation. In fact, building a proper model is costly, hard, and requires both mathematical skills and ingenuity. Moreover, even if models are developed, they are often not updated with the changes in the implementation and therefore the models and the implementation progressively diverge.

Model inference is a promising approach to tackle this problem by using machine learning to infer software behavior models automatically from execution logs [1–3]. Many model inference algorithms [4–6] have been proposed by recent research. To infer accurate models, the logs should contain as much detail information as possible. However, a log with more information also increases the difficulty of model inference task. The logs generated by real systems are usually very

* Corresponding author.
E-mail addresses: cluo8@uci.edu (C. Luo), hefei@tsinghua.edu.cn (F. He), carlo.ghezzi@polimi.it (C. Ghezzi).

[1] The work was done while the author was at Tsinghua University. The author is now at University of California, Irvine, USA.

http://dx.doi.org/10.1016/j.scico.2017.04.004
0167-6423/

large. For example, Prospex [7] infers state machines from network logs for vulnerability analysis of network applications. In practice, network logs collected passively can be enormous, while models need to be inferred quickly to ensure timeliness of subsequent analyses.

A serious problem with existing model inference methods [1–3,8,5,6] is that they do not scale up to very large logs, since they are all centralized. Recent work [9] has shown that many existing model inference algorithms run out of memory or take hours to complete when processing large logs. It is thus desirable to parallelize the processing of massive logs in model inference tasks. Many existing model inference methods share a similar workflow. They first slice the input log into a set of log pieces based on some criteria, each of which constitutes a run of the system. The system model is then synthesized from the sliced log pieces together. Thus an interesting research question is to devise a *generalized scalable* approach to model inference exploiting the potential benefits of state-of-the-art distributed processing infrastructures.

Previous work by Lee et al. [8] proposes a general algorithm to slice logs by parametric events, which is a useful but expensive step in model inference. A natural idea to parallelize this algorithm is to divide the log into multiple segments, slice each segment on one node, and then merge the sliced results. However, this naive solution could lead to incorrect results since events in different segments may be correlated and should be processed together (Section 4.2). To handle this, we propose a distributed slicing algorithm. Using the MapReduce model [10], we can effectively distribute the slicing of massive logs to numerous computing nodes, meanwhile ensuring that the correlated events are always properly processed.

With the set of traces obtained by slicing the log, many off-the-shelf model synthesis algorithms [1,2,11] can be applied to infer the system model. However, since these are centralized algorithms and the traces can be very large, we further propose a distributed model synthesis algorithm based on k-tail [1] to improve scalability. With the powerful data processing capacity of MapReduce, the problem of inferring behavior models from large logs can be efficiently solved.

In a nutshell, our approach consists of two phases: *trace slicing* and *model synthesis*. The first phase parses and slices the log into a set of trace slices and constructs a prefix tree acceptor (PTA) [4], which is a compact structure to store a set of traces. The second phase then reads the PTA and synthesizes the behavior model. We develop a distributed algorithm for the trace slicing and model synthesis phases, respectively. With these two algorithms, we propose a novel framework for inferring software behavior models with MapReduce. Note that the two phases in our framework are designed to be decoupled. This design scheme allows users to plug existing centralized model synthesis algorithms into our framework to infer more accurate models (Section 8.2).

The main contributions are summarized as follows:

- We propose a distributed trace slicing algorithm using MapReduce;
- We propose a distributed model synthesis algorithm using MapReduce;
- With above algorithms, we developed an inference approach that, to the best of our knowledge, represents a novel attempt to infer software behavior models with MapReduce;
- We implemented a prototype of our technique. The experimental results show the promising performance of our approach.

This paper is based on our previous work [12] and presents several extensions. First, we describe several practical optimizations to further improve our approach. Secondly, we formally prove correctness of our distributed trace slicing and distributed model synthesis algorithms respectively. Finally, we provide more complete experimental assessment of our approach under various settings.

The rest of the paper is organized as follows: Section 2 provides an overview of our approach. Section 3 introduces preliminary concepts. Section 5 and Section 6 introduce our distributed algorithms for trace slicing and model synthesis, respectively. Section 7 reports the experimental results. Section 8 describes some possible extensions. Section 9 discusses the related work and Section 10 concludes this paper.

2. Overview

2.1. MapReduce

MapReduce [10] is a large-scale parallel data processing framework supported by a distributed architecture. It hides the details of data distribution, load balancing, replication, and also scheduling, while provides simple yet powerful interfaces to users. Due to its simplicity, MapReduce has become one of the most popular distributed computing frameworks. Hadoop[2] is a popular open-source implementation of MapReduce.

In MapReduce, the data is stored in a distributed file system (DFS). The computation is based on key–value pairs and expressed via the following two functions:

$$\textsc{Map}: \quad (k1, v1) \rightarrow list(k2, v2)$$

$$\textsc{Reduce}: \quad (k2, list(v2)) \rightarrow list(k3, v3)$$

[2] http://hadoop.apache.org/.

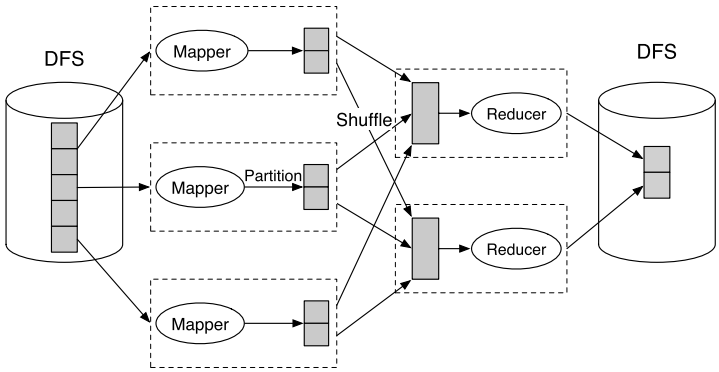


Fig. 1. MapReduce overview.

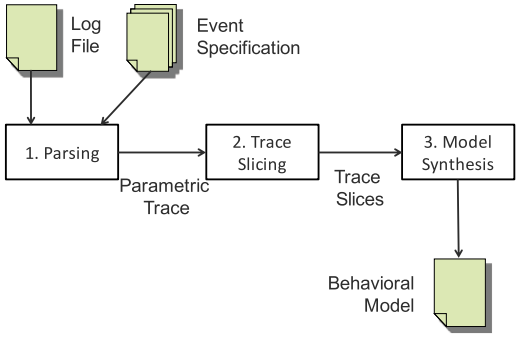


Fig. 2. Behavioral model inference overview.

The processing of a dataset, i.e., the execution of a map and reduce function, is expressed as a MapReduce job, whose main flow is illustrated in Fig. 1. The job consists of three phases, i.e., *map*, *shuffle*, and *reduce*. In the *map* phase, the input data is partitioned and distributed to a number of mappers. At each mapper, a user-defined Map function is invoked to handle the input data and produce intermediate results (in the form of key–value pairs). These intermediate results are then partitioned and sorted by their keys in the *shuffle* phase. Each partition is processed by a reducer in the *reduce* phase. At each reducer, a user-defined Reduce function is invoked to handle that partition. Note that the MapReduce framework ensures the values for the same key are passed to a single reduce call. The output of a reducer is written to the DFS.

Besides the Map and Reduce functions, the user can optionally define the Combine function. This function works as a local reducer in each mapper by reducing the amount of intermediate key–value pairs sent across the network. The MapReduce framework also allows users to provide Initialization and Teardown functions for each mapper and reducer, and to customize the Partition and Comparison functions used for partitioning and sorting the key–value pairs during the shuffle phase. These mechanisms provide more flexibility for users designing algorithms with MapReduce.

When solving a problem on top of MapReduce, one major concern is to design the distributed algorithm with the Map and Reduce functions. Once the algorithm is well encoded, one can leverage clusters and parallel computing to speed up the computation. The interested reader may refer to [10,13] for more information.

2.2. Behavioral model inference

The workflow of a typical model inference approach is shown in Fig. 2, which consists of three steps: *log parsing*, *trace slicing*, and *model synthesis*. In the first step, we rely on a parser to extract relevant events from the log files. The relevant events are defined by the *event specification*. The events are usually associated with some parameters, called *parametric events*. After parsing, we get a sequence of parametric events, called a *parametric trace*.

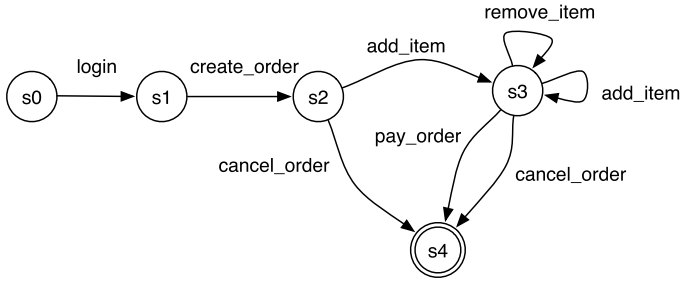
In general, the parametric trace cannot be used as it is to synthesize the system model directly as it contains many independent and interleaved runs. Thus, a trace slicer is called to slice the parametric trace into slices. Each parametric trace has the same combination of parameters, and corresponds to a run of the system. Finally, a synthesis algorithm is called to infer the behavior model from the set of trace slices. Typically, a synthesis algorithm first represents the trace slices as a prefix tree acceptor (PTA) [4] as the initial model, which accepts the set of trace slices exactly. Then, the initial model is generalized by iteratively merging equivalent states based on certain criteria to produce a more compact but general one.

Events	Parameters
login	userid
create_order	userid, orderid
add_item	orderid, itemid
remove_item	orderid, itemid
pay_order	userid, orderid
cancel_order	userid, orderid

(a) Parametric event definition

```
1:  login (user1)
2:  create_order(user1,order1)
3:  login (user2)
4:  create_order(user2,order2)
5:  add_item (order1,item1)
6:  add_item (order2,item2)
7:  remove_item (order2,item2)
8:  pay_order (user1,order1)
9:  cancel_order(user2,order2)
```

(b) A parametric trace



(c) The behavior model

Fig. 3. An online shopping system example.

2.3. Running example

As a running example, consider the on-line shopping system shown in Fig. 3. The relevant events and their corresponding parameters are shown in Fig. 3a. Briefly, we are interested in the following events:

- the user *userid logs in* in the system,
- the user *userid creates* an order with the ID of *orderid*,
- the item *itemid* is *added* to the order *orderid*,
- the item *itemid* is *removed* from the order *orderid*,
- the user *userid pays* the order *orderid*, and
- the user *userid cancels* the order *orderid*.

Note that there may be more parameters for each event than the ones we listed. For example, to create an order, more information (like the creation time, etc.) may be recorded in the log file, but only the *userid* and *orderid* are assumed to be relevant in our case.

The behavior model of the online shopping system is depicted in Fig. 3c. Notice, however, that we assume that the model is initially *unknown* to us. Our goal is exactly to infer the behavior model from lots of log files generated by the system. Note that these logs may contain a lot of information, such as events and parameters, which is irrelevant to the model inference task. Thus we need a parser to extract relevant events and parameters from a log file. A parametric trace excerpt is shown in Fig. 3b. Since multiple users can operate in the shopping system at the same time, the events corresponding to their operations are interleaved in the log file.

2.4. Our approach

The log file may be too large to be managed by existing model inference algorithms on a single machine. To deal with this problem, we propose to apply MapReduce to parallelize the model inference task.

As shown in Fig. 4, our approach consists of two phases, i.e., the distributed trace slicing phase and the distributed model synthesis phase, both of which are realized using MapReduce. The first phase takes as input a log file, performs the log parsing and trace slicing, and outputs a prefix tree acceptor (PTA). The log parsing task is performed by mappers, while the trace slicing task is executed by reducers. Both tasks are distributed (implicitly by the MapReduce architecture) to a number of computing nodes. The second phase takes as input the PTA generated in the former phase, and outputs the behavior model by a distributed model synthesis algorithm.

Although the basic algorithms for trace slicing [8] and model synthesis [1] exist, our contribution is to show how they can be cast and integrated into a novel scalable distributed framework based on MapReduce.

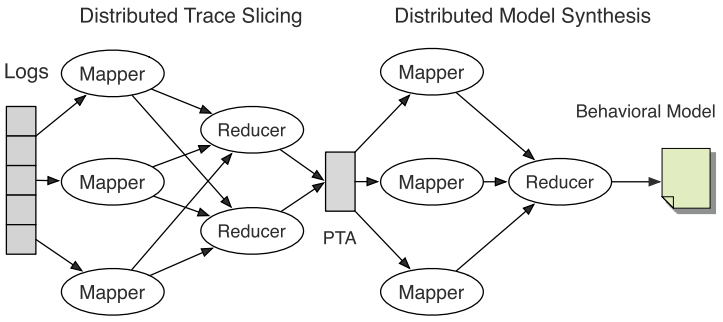


Fig. 4. Model inference with MapReduce.

3. Formal definitions

This section introduces the formal definitions needed in our framework. Some of these definitions originate from [8].

Definition 1. An *event specification* is a pair $\langle \mathcal{E}, X \rangle$, where $\mathcal{E}$ is a set of *base events*, and X is a set of parameters.

An event specification specifies the events and the parameters of interest. For example, the event specification in Fig. 3a is $\mathcal{E} = \{login, create\_order, add\_item, remove\_item, pay\_order, cancel\_order\}$, $X = \{userid, orderid, itemid\}$.

Let $[A \to B]$ (or $[A \rightharpoonup B]$) be the set of total (or partial) functions from A to B. For any partial function $\theta \in [A \rightharpoonup B]$, $Dom(\theta) = \{x \in A \mid \theta(x) \text{ is defined}\}$. Let $\perp$ be the partial function for which $Dom(\perp) = \emptyset$.

Definition 2. A *parameter instance* θ is a partial function from X to V_X, i.e., $\theta \in [X \rightharpoonup V_X]$, where V_X is a set of parameter values for the parameter set X. A parameter instance θ is said to be *complete* if $Dom(\theta) = X$. Let $Y \subseteq Dom(\theta)$, a restriction $\theta \upharpoonright_Y$ of θ to Y is a parameter instance such that $Dom(\theta \upharpoonright_Y) = Y$ and for any $y \in Y$, $\theta \upharpoonright_Y (y) = \theta(y)$.

To simplify the notation, we often ignore X and use V_X to represent the parameter instance, if X and the mapping from X to V_X is clear from the context. For example, the partial function $\langle userid \mapsto user1, orderid \mapsto order1 \rangle$ is a parameter instance for the running example, which can be abbreviated as $\langle user1, order1 \rangle$.

Definition 3. The *parametric event definition* $\mathcal{D}_e$ is a function from $\mathcal{E}$ to 2^X, i.e., $\mathcal{D}_e \in [\mathcal{E} \to 2^X]$. A *parametric event* is $e\langle\theta\rangle$, where e is a base event, θ is a parameter instance such that $Dom(\theta) = \mathcal{D}_e(e)$.

A parametric event definition provides parameter information for each base event $e \in \mathcal{E}$, and we assume a base event must have same parameters (not necessarily same values) in every position of the log where it occurs [8].

Definition 4. A parameter instance θ' is said to be less or equal informative than another parameter instance θ (written $\theta' \sqsubseteq \theta$), if for any $x \in X$, $\theta'(x)$ is defined implies $\theta(x)$ is also defined and $\theta'(x) = \theta(x)$.

Consider the running example (Fig. 3b). The parameter instance $\langle user1 \rangle$ is less or equal informative than $\langle user1, order1 \rangle$, and $\langle user1, order1 \rangle$ is less or equal informative than itself.

Definition 5. A *trace* is a finite sequence of base events. A *parametric trace* is a finite sequence of parametric events. We write $e \in \tau$ (or $e\langle\theta\rangle \in \tau$) if the base event e (or the parametric event $e\langle\theta\rangle$) appears in the trace (or the parametric trace) τ.

For example, each line in Fig. 3b represents a parametric event, and the sequence of all parametric events in Fig. 3b represents a parametric trace.

Definition 6. Let τ be a parametric trace, and θ be a parameter instance, the *θ-trace slice* $\tau \upharpoonright_\theta$ of τ is a non-parametric trace recursively defined as:

- $\epsilon \upharpoonright_\theta = \epsilon$, where ϵ is the empty trace, and
- $(\tau e\langle\theta'\rangle) \upharpoonright_\theta = \begin{cases} (\tau \upharpoonright_\theta) e, & \text{if } \theta' \sqsubseteq \theta \\ \tau \upharpoonright_\theta, & \text{otherwise} \end{cases}$

Intuitively, $\tau\restriction_\theta$ filters out irrelevant parametric events in τ with respect to θ. Specially, the θ-trace slice of an empty trace ϵ is empty. If the trace to be sliced is not empty, denoting as $\tau e\langle\theta'\rangle$, where $e\langle\theta'\rangle$ is the ending parametric event of this trace, the base event e is kept in the slice only if $\theta' \sqsubseteq \theta$. The leading part τ is then sliced in a recursive way.

For example, let τ_1 be the parametric trace in Fig. 3b. For parameter instance $\theta_1 = \langle user1, order1\rangle$, $\tau_1\restriction_{\theta_1}$ is the sequence of: *login, create_order, pay_order*. For another parameter instance $\theta_2 = \langle user1, order1, item1\rangle$, $\tau_1\restriction_{\theta_2}$ is the sequence of: *login, create_order, add_item, pay_order*.

A trace slice corresponds to a parameter instance. However, as we can see, all parameter instances appearing in τ_1 are incomplete. With the following operators, some incomplete parameter instances can be combined to form a complete one.

Definition 7. Two parameter instances θ and θ' are *compatible* if for any $x \in Dom(\theta) \cap Dom(\theta')$, $\theta(x) = \theta'(x)$. If θ and θ' are compatible, we define their *combination* (written $\theta \sqcup \theta'$) as:

$$(\theta \sqcup \theta')(x) = \begin{cases} \theta(x) & \text{if } \theta(x) \text{ is defined} \\ \theta'(x) & \text{if } \theta'(x) \text{ is defined} \\ undefined & \text{otherwise} \end{cases}$$

Consider the running example (Fig. 3b). The parameter instances $\langle user1, order1\rangle$ and $\langle order1, item1\rangle$ are compatible, while their combination gives $\langle user1, order1, item1\rangle$, and their intersection gives $\langle order1\rangle$. However, the parameter instances $\langle user1\rangle$ and $\langle user2, order2\rangle$ are incompatible.

The combination of parameter instances may lead to meaningless results. For example, the parameter instance $\langle user1\rangle$ and $\langle order2, item2\rangle$ are compatible, but their combination $\langle user1, order2, item2\rangle$ is meaningless since *user*1 and *order*2 do not interact in any event. To avoid such meaningless combinations, we require only *connected* parameter instances to be combined.

Definition 8. Given two parameter instances θ_1 and θ_2, we say θ_1 and θ_2 are *strongly compatible* (written $\theta_1 \bowtie \theta_2$), if θ_1 and θ_2 are compatible, and $Dom(\theta_1) \cap Dom(\theta_2) \neq \emptyset$.

Definition 9. Given a parametric trace τ and a parameter instance θ, we say θ is τ*-connected* (or *connected* if τ is clear from the context), if

- there exists e such that $e\langle\theta\rangle \in \tau$, or
- there exist θ_1 and θ_2 such that both θ_1 and θ_2 are τ-connected, $\theta_1 \bowtie \theta_2$, and $\theta = \theta_1 \sqcup \theta_2$.

Consider the running example (Fig. 3b). The parameter instances $\langle user1, order1\rangle$ and $\langle order1, item1\rangle$ satisfy the first condition in the above definition, and are thus connected. By Definition 8, these two parameter instances are strongly compatible. Moreover, $\langle user1, order1\rangle \sqcup \langle order1, item1\rangle = \langle user1, order1, item1\rangle$, the parameter instance $\langle user1, order1, item1\rangle$ is thus also connected.

In the remainder of this paper, we consider only trace slices for complete and connected parameter instances to avoid meaningless results, as in [8].

4. Sequential trace slicing

In this section we briefly review the sequential trace slicing algorithm proposed by Lee et al. [8]. Then we discuss a naive parallelization strategy based on this algorithm.

4.1. The algorithm

The pseudo code of the sequential trace slicing algorithm is shown in Fig. 5, where the function Slice takes as input a parametric trace τ, and produces a set of trace slices as output.

At lines 2 to 5, the algorithm sorts the parametric events in τ into several lists. Let Δ be the set of these event lists. Each list $\Delta(\theta)$ corresponds to a parametric instance θ and contains all base events e such that $e\langle\theta\rangle$ occurs in τ. $Dom(\Delta)$ thus represents the set of parametric instances occurring in τ, i.e., $Dom(\Delta) = \{\theta | \exists e.e\langle\theta\rangle \in \tau\}$. Assume that each event is associated with a *timestamp*. At line 5, the base event e is inserted in a proper way such that all events in $\Delta(\theta)$ are in ascending order of their timestamps.

At line 6, the algorithm initializes the set Ω to $Dom(\Delta)$. At line 7, the algorithm tries to find any two parametric instances in Ω, such that they are strongly compatible and their combination is not currently in Ω. If such two parametric instances exist, their combination is added (at line 8) to Ω, and the above process is iterated (since the set Ω has been changed). With the operation at line 6 and the operations at lines 7 to 8, the connected parametric instances satisfying the first and the second condition of Definition 9 are added to Ω, respectively. Finally, the set Ω contains (and exactly contains) all connected parametric instances.

```
1: function SLICE(τ)
2:     for e⟨θ⟩ ∈ τ do
3:         if θ ∉ Dom(Δ) then
4:             Initialize Δ(θ) as an empty list;
5:         Insert e into Δ(θ);
6:     Ω ← Dom(Δ);
7:     while ∃θ1, θ2 ∈ Ω s.t. θ1 ⋈ θ2, (θ1 ⊔ θ2 ∉ Ω) do
8:         Ω ← Ω ∪ {θ1 ⊔ θ2};
9:     for complete θ ∈ Ω do
10:        Γ ← {Δ(θ′)|θ′ ⊑ θ, θ′ ∈ Dom(Δ)};
11:        τ ↾θ ← merging event lists in Γ;
12:        Output τ ↾θ
```

Fig. 5. Sequential trace slicing algorithm.

At lines 9 to 12, the algorithm constructs a trace slice $\tau \restriction_\theta$ for each complete parametric instance θ in Ω. Let θ' be any parametric instance that occurs in τ and is less or equal informative than θ. The $\tau \restriction_\theta$ is constructed by merging all $\Delta(\theta')$'s. The merge operation is required to keep the ascending order of events on their timestamps.

For example, let us apply the SLICE function to the parametric trace τ_1 in Fig. 3b. After the first for loop (lines 2 to 5), Δ is set to:

$$\Delta(\langle user1\rangle) = login$$

$$\Delta(\langle user1, order1\rangle) = create\_order, pay\_order$$

$$\Delta(\langle user2\rangle) = login$$

$$\Delta(\langle user2, order2\rangle) = create\_order, cancel\_order$$

$$\Delta(\langle order1, item1\rangle) = add\_item$$

$$\Delta(\langle order2, item2\rangle) = add\_item, remove\_item$$

At line 6 Ω is assigned the set $Dom(\Delta) = \{\langle user1\rangle, \langle user1, order1\rangle, \langle user2\rangle, \langle user2, order2\rangle, \langle order1, item1\rangle, \langle order2, item2\rangle\}$. The while loop at lines 7 to 8 adds new parametric instances $\theta_1 = \langle user1, order1, item1\rangle$ and $\theta_2 = \langle user2, order2, item2\rangle$ to Ω. Note that only θ_1 and θ_2 in Ω are complete. The final for loop (lines 9 to 12) constructs the trace slices for θ_1 and θ_2:

$$\tau_1 \restriction_{\theta_1} = login, create\_order, add\_item, pay\_order$$

$$\tau_1 \restriction_{\theta_2} = login, create\_order, add\_item, remove\_item, cancel\_order$$

4.2. A naive parallelization strategy

As mentioned before, a naive strategy to parallelize the trace slicing task divides the trace into multiple segments, processes each segment with the above sequential slicing algorithm, and then merges the sliced results together. However, this strategy could lead to incorrect results.

For example, suppose the trace τ_1 in Fig. 3b is divided into two segments τ_1^1 and τ_1^2, where τ_1^1 contains the events from 1 to 4, while τ_1^2 contains the rest. We first process τ_1^1 using the SLICE function. At line 9, Ω for τ_1^1 is

$$\Omega_1^1 = \{\langle user1\rangle, \langle user1, order1\rangle, \langle user2\rangle, \langle user2, order2\rangle\}.$$

According to Definition 9, there is no complete and connected parametric instance in Ω_1^1. Thus no slice is generated for τ_1^1.

Then we process τ_1^2. At line 9 of the SLICE function, Ω for τ_1^2 is

$$\Omega_1^2 = \{\langle order1, item1\rangle, \langle order2, item2\rangle, \langle user1, order1\rangle, \langle user2, order2\rangle,$$
$$\langle user1, order1, item1\rangle, \langle user2, order2, item2\rangle\}.$$

The last two parametric instances in Ω_1^2 (named as θ_1 and θ_2, respectively) are complete. The trace slices of τ_1^2 for θ_1 and θ_2 are:

$$\tau_1^2 \restriction_{\theta_1} = add\_item, pay\_order$$

$$\tau_1^2 \restriction_{\theta_2} = add\_item, remove\_item, cancel\_order$$

Merging trace slices of τ_1^1 and τ_1^2 also gives $\tau_1^2 \restriction_{\theta_1}$ and $\tau_1^2 \restriction_{\theta_2}$. These results are clearly incorrect.

The naive solution could lead to incorrect results since the parametric events in different segments (for example, $create\_order(user1, order1)$ in τ_1^1 and $add\_item(order1, item1)$ in τ_1^2) may be correlated and should be processed together.

5. Distributed trace slicing with MapReduce

In this section, we introduce our distributed trace slicing approach with MapReduce. We first propose a data encoding mechanism, and then introduce the MAP and REDUCE functions. We also discuss several practical optimizations to our approach.

5.1. Data encoding

In MapReduce, the transmitted data between mappers and reducers are organized as key–value pairs. The transmitted data for our problem are basically parametric events. We thus need a mechanism to set a *key* for each parametric event to distribute them to reducers.

The basic idea is to first choose a subset $\mathcal{X}$ of X, and for each parametric event $e\langle\theta\rangle$, we report the values of $\mathcal{X}$ on parametric instance θ as its key. The key is then used by MapReduce to determine the reducer to which the parametric event should be passed.

Definition 10. A *parameter window* $\mathcal{X}$ is a subset of X, such that for all $e \in \mathcal{E}$, either $\mathcal{X} \subseteq \mathcal{D}_e(e)$ or $\mathcal{X} \cap \mathcal{D}_e(e) = \emptyset$. A parameter window $\mathcal{X}$ is *nontrivial* if $\mathcal{X} \neq \emptyset$.

Note that any singleton parameter set is always a well-formed and nontrivial parameter window. For the running example, a nontrivial parameter window can be $\mathcal{X} = \{orderid\}$.

Definition 11. The *key* of a parametric event $e\langle\theta\rangle$ (written $key(e\langle\theta\rangle)$) with respect to the parameter window $\mathcal{X}$ is

- the restriction of θ to $\mathcal{X}$, i.e., $\theta\restriction_{\mathcal{X}}$, if $\mathcal{X} \subseteq \mathcal{D}_e(e)$, or
- $\perp$, if $\mathcal{X} \cap \mathcal{D}_e(e) = \emptyset$.

For example, with the parameter window $\mathcal{X} = \{orderid\}$, the key of the first parametric event $login\langle user1\rangle$ in Fig. 3b is $\perp$. And the keys of the remaining parametric events in Fig. 3b are: $\langle order1\rangle$, $\perp$, $\langle order2\rangle$, $\langle order1\rangle$, $\langle order2\rangle$, $\langle order2\rangle$, $\langle order1\rangle$ and $\langle order2\rangle$, respectively.

With a parameter window $\mathcal{X}$, we divide all parametric events in a trace into two disjoint sets: $T_1 = \{e\langle\theta\rangle | \mathcal{X} \subseteq \mathcal{D}_e(e)\}$ and $T_2 = \{e\langle\theta\rangle | \mathcal{X} \cap \mathcal{D}_e(e) = \emptyset\}$. Continuing the previous example, the parametric events labeled 2, 4, 5, 6, 7, 8, and 9 belong to T_1, and the remaining parametric events belong to T_2.

Lemma 1. *Let $e_1\langle\theta_1\rangle$ and $e_2\langle\theta_2\rangle$ be two parametric events in T_1 such that $key(e_1\langle\theta_1\rangle) \neq key(e_2\langle\theta_2\rangle)$, then $e_1\langle\theta_1\rangle$ and $e_2\langle\theta_2\rangle$ must be* incompatible.

Proof. Since the two parametric events belong to T_1, $\mathcal{X} \subseteq \mathcal{D}_e(e_1) \cap \mathcal{D}_e(e_2)$. Moreover, as $key(e_1\langle\theta_1\rangle) \neq key(e_2\langle\theta_2\rangle)$, there must exist $x \in \mathcal{X} \subseteq \mathcal{D}_e(e_1) \cap \mathcal{D}_e(e_2)$ such that $\theta_1(x) \neq \theta_2(x)$. Thus the statement holds. □

Let $hash()$ be a hash function that takes a key as input and returns the ID of a reducer. For a parametric event $e_1\langle\theta_1\rangle \in T_1$, let $k_1 = key(e_1\langle\theta_1\rangle)$, we pass the key–value pair $(k_1, e_1\langle\theta_1\rangle)$ to the reducer whose ID is $hash(k_1)$. However, parametric events in T_2 may be combined with any parametric events in T_1. Thus, for any parametric event $e_2\langle\theta_2\rangle \in T_2$, we pass the key–value pair $(\perp, e_2\langle\theta_2\rangle)$ to all reducers.

Consider the running example (Fig. 3b) with $\mathcal{X} = \{order\}$, and assume $hash(\langle order1\rangle) = 1$ and $hash(\langle order2\rangle) = 2$. Then the parametric events labeled 2, 5 and 8 in T_1 are passed to $Reducer_1$, the parametric events labeled 4, 6 7 and 9 in T_1 are passed to $Reducer_2$. The parametric events labeled 1 and 3 in T_2 are passed to both reducers.

5.1.1. Choosing the parameter window

We now discuss how to choose the parameter window $\mathcal{X}$ automatically. Since parametric events in T_2 need to be passed to all reducers, $\mathcal{X}$ should be chosen such that T_2 is as small as possible. However, the optimal $\mathcal{X}$ cannot be determined unless we have processed the entire log. To handle this, we define the non-parametric version of T_2 as $\hat{T_2} = \{e | \mathcal{X} \cap \mathcal{D}_e(e) = \emptyset\}$, and relax the criteria as follows:

Heuristics 1. The set $\mathcal{X}$ should be chosen such that $\hat{T_2}$ is as small as possible.

This heuristics is an approximation, since minimizing $\hat{T_2}$ does not necessarily mean that T_2 is minimized. However, one advantage is that $\hat{T_2}$ can be computed with the event definitions, which is known a priori. Thus, the parameter window $\mathcal{X}$ can be decided before MapReduce computations.

Consider the running example in Fig. 3. According to Definition 10, all non-trivial parameter windows are $\{userid\}$, $\{orderid\}$ and $\{itemid\}$. $\hat{T_2}$ with respect to a parameter window can be computed by looking at the *Parameters* column of

```
1: function MAP(line)
2:     e⟨θ⟩ ← PARSE(line);
3:     if e⟨θ⟩ = NULL then
4:         return ;
5:     if 𝒳 ⊆ 𝒟_e(e) then
6:         OUTPUT(θ↾𝒳, e⟨θ⟩);
7:     else
8:         OUTPUT(⊥, e⟨θ⟩);
```

Fig. 6. Trace slicing: MAP function.

Fig. 3a, and finding out the events whose parameters are disjoint with this window. As a result, $\hat{T}_2$ with respect to $\{userid\}$, $\{orderid\}$ and $\{itemid\}$ are $\{add\_item, remove\_item\}$, $\{login\}$ and $\{login, create\_order, pay\_order, cancel\_order\}$, respectively. According to Heuristics 1, we would choose the parameter window $\{orderid\}$.

Moreover, for parametric events in T_1, we want them to be distributed evenly to reducers. In other words, we want keys in T_1 to be as many as possible. Notice that the number of different keys is influenced by $|\mathcal{X}|$, we thus have another heuristics.

Heuristics 2. The set $\mathcal{X}$ should be as large as possible.

With above heuristics, the parameter window $\mathcal{X}$ can be decided with a brute-force search as follows. We first find all non-trivial parameter windows according to Definition 10, then apply the first to minimize $\hat{T}_2$. If there are multiple candidates $\mathcal{X}$, we then apply the second heuristics to select the one with the largest size.

5.2. Mapper

The log is split (implicitly by the MapReduce) into blocks, each of which is passed to a mapper. We call each line in the log a *log entry*. A log entry records a parametric event, and the time when it happens. In the remainder of the paper, we assume each event to be associated with a *timestamp*. However, for simplicity, we will consider them only when we need to sort the parametric events.

Fig. 6 shows the pseudocode of the MAP function, which takes as input a log entry and outputs a key–value pair. Note that the parameter window $\mathcal{X}$ is provided a priori to all mappers. For each log entry, the PARSE function is called (line 2) to get the parametric event $e\langle\theta\rangle$. If the event is not in $\mathcal{E}$, the PARSE function returns NULL and this log entry is simply skipped (line 4). Otherwise, the mapper outputs a key–value pair (lines 5–8) based on Definition 11, where the key is $\theta\restriction_{\mathcal{X}}$ if $\mathcal{X} \subseteq \mathcal{D}_e(e)$ and $\perp$ otherwise, and the value is the parametric event itself.

Consider the running example (Fig. 3b) with $\mathcal{X} = \{orderid\}$, and assume there are two mappers. Then the parametric events labeled from 1 to 5 in Fig. 3b are handled by $Mapper_1$, and the parametric events labeled from 6 to 9 in Fig. 3b are handled by $Mapper_2$. After processing the parametric events, $Mapper_1$ outputs the following key–value pairs:

$$1 : (\perp, login\langle user1\rangle)$$

$$2 : (\langle order1\rangle, create\_order\langle user1, order1\rangle)$$

$$3 : (\perp, login\langle user2\rangle)$$

$$4 : (\langle order2\rangle, create\_order\langle user2, order2\rangle)$$

$$5 : (\langle order1\rangle, add\_item\langle order1, item1\rangle)$$

$Mapper_2$ outputs the following key–value pairs:

$$6 : (\langle order2\rangle, add\_item\langle order2, item2\rangle)$$

$$7 : (\langle order2\rangle, remove\_item\langle order2, item2\rangle)$$

$$8 : (\langle order1\rangle, pay\_order\langle user1, order1\rangle)$$

$$9 : (\langle order2\rangle, cancel\_order\langle user2, order2\rangle)$$

For convenience of the representation, we label above key–value pairs with the same label as the parametric events. Assume $hash(\langle order1\rangle) = 1$ and $hash(\langle order2\rangle) = 2$. The distribution of Mappers' outputs to reducers is shown in Fig. 7. For example, the key–value pairs labeled 2 and 5 belong to T_1, and thus are passed to $Reducer_1$; the key–value pairs labeled 1 and 3 belong to T_2, and are passed to both reducers.

5.3. Reducer

Recall that in the shuffle phase, MapReduce partitions and sorts key–value pairs to ensure that values corresponding to the same key are organized into the same list. Specifically, the parametric events in T_1 are sorted into multiple

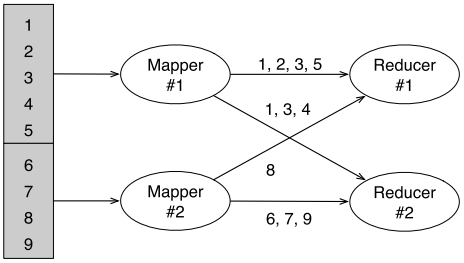


Fig. 7. Trace slicing: the running example.

```
1: function REDUCE(key, values[])
2:     if key = ⊥ then
3:         Δ⊥ ←RESTORE(values[]);
4:         return ;
5:     Δ ←RESTORE(values[]);
6:     while ∃θ⊥ ∈ Dom(Δ⊥), ∃θ ∈ Dom(Δ)
7:       s.t. θ⊥ ∉ Dom(Δ) ∧ θ⊥ ⋈ θ do
8:         Δ(θ⊥) ← Δ⊥(θ⊥);
9:     CONSTRUCT(Δ);
10: function RESTORE(values[])
11:     Δ ← ∅;
12:     for e⟨θ⟩ ∈ values[] do
13:         if θ ∉ Dom(Δ) then
14:             Initialize Δ(θ) as an empty list;
15:         Insert e into Δ(θ);
16:     return Δ;
17: function CONSTRUCT(Δ)
18:     Ω ← Dom(Δ);
19:     while ∃θ1, θ2 ∈ Ω
20:       s.t. θ1 ⋈ θ2, (θ1 ⊔ θ2 ∉ Ω) do
21:         Ω ← Ω ∪ {θ1 ⊔ θ2};
22:     for complete θ ∈ Ω do
23:         Γ ← {Δ(θ′)|θ′ ⊑ θ, θ′ ∈ Dom(Δ)};
24:         τ ↾θ ← merging event lists in Γ;
25:         Update PTA using τ ↾θ;
```

Fig. 8. Distributed trace slicing: REDUCE function.

key–value pairs, each with a distinct key; and the parametric events in T_2 are sorted into the same key–value pair with $key = \perp$. Let $values[]$ denote the list of parametric events with the key key. The REDUCE function is called for each pair $(key, values[])$. Consider $Reducer_1$ in Fig. 7, and let $key = \langle order1 \rangle$. Then $values[] = create\_order\langle user1, order1\rangle$, $add\_item\langle order1, item\rangle$, $pay\_order\langle user1, order1\rangle$.

The REDUCE function is shown in Fig. 8. To ease the possible combinations of parametric events between T_1 and T_2, the MapReduce framework is configured such that the key–value pair corresponding to T_2 always come first. The RESTORE function acts in the same way as the lines 2 to 5 of the SLICE function (Fig. 5). The parametric events in $values[]$ are sorted into several lists. Each list $\Delta(\theta)$ corresponds to a parametric instance θ. Since the key–value pair with $key = \perp$ always comes first at each reducer, when the REDUCE function proceeds to line 5, $\Delta_\perp$ must have already been initialized. The while loop at line 6 tries to copy the list $\Delta_\perp(\theta_\perp)$ in $\Delta_\perp$ to Δ, if the parametric instance $\theta_\perp$ does not exist in $Dom(\Delta)$ and is strong compatible with some parametric instance θ in $Dom(\Delta)$. Note that $\theta_\perp$ may again be strongly compatible with other parametric instances in T_2, this process is thus iterative. At line 9, the CONSTRUCT function is called to compute trace slices and update the intermediate structure PTA. The CONSTRUCT function acts almost identical to the corresponding part (lines 6 to 12 in Fig. 5) of the sequential slicing algorithm, except for the last statement.

To illustrate the algorithm, consider $Reducer_1$ of our running example. After line 5 of the REDUCE function, $\Delta_\perp$ and Δ are as follows:

$$\Delta_\perp(\langle user1\rangle) = login$$

$$\Delta_\perp(\langle user2\rangle) = login$$

$$\Delta(\langle user1, order1\rangle) = create\_order, pay\_order$$

$$\Delta(\langle order1, item1\rangle) = add\_item$$

At line 6, since $\langle user1\rangle$ is strongly compatible with $\langle user1, order1\rangle$, the list $\Delta_\perp(\langle user1\rangle)$ is added to Δ. Then after the while loop at line 19, $\Omega = \{\langle user1\rangle, \langle user1, order1\rangle, \langle order1, item1\rangle, \langle user1, order1, item1\rangle\}$. Note that the last instance in Ω is a combined instance and is also complete. Let $\theta = \langle user1, order1, item1\rangle$, then $\tau \upharpoonright_\theta = login, create\_order, add\_item, pay\_order$.

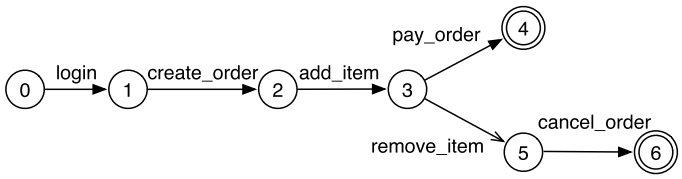


Fig. 9. PTA for the running example.

We take the prefix tree acceptor (PTA) as the intermediate structure. Each reducer keeps a PTA. The generated trace slices (at line 25) are used to iteratively update the PTA. Note that the PTA maintained at each reducer is partial, i.e., it only accepts trace slices generated at the reducer. However, since the model inference algorithm (see Section 6) takes as input a complete PTA, we then merge the PTAs in each reducer to form a complete one after the reduce process terminates. The complete PTA accepts all trace slices generated on all reducers. The PTA for the running example is shown in Fig. 9.

Correctness. The correctness of our distributed trace slicing algorithm is ensured by the following theorem.[3]

Theorem 1. *Given a parametric trace τ, the following statements hold for the distributed trace slicing job:*

1. *for any $\tau\restriction_\theta$ constructed at line 24, the parametric instance θ is complete and connected, and*
2. *for any complete and connected parametric instance θ, θ-trace slice of τ is constructed at line 24.*

Intuitively, the first statement ensures that only trace slices for those complete and connected parametric instances are generated, while the second statement guarantees that trace slices for all complete and connected parametric instances are generated. The two statements ensure soundness and completeness of our algorithm, respectively.

5.4. Optimizations

We have developed several optimizations to improve our distributed trace slicing algorithm.

COMBINE function for mappers. We implemented a COMBINE function, which performs local reduce operations to reduce the intermediate key–value pairs output by each mapper. Before passing the key–value pairs to reducers, the MapReduce framework calls the COMBINE function to merge the set of base events with the same parametric instance into a list. Thus, the COMBINE function helps to reduce the size of intermediate key–value pairs, which in turn lowers the time of the shuffling phase.

Partitioning parametric instances. Let θ_1 and θ_2 be two parametric instances such that $Dom(\theta_1) \subseteq Dom(\theta_2)$, either θ_1 and θ_2 are not compatible, or their combination generates no new parametric instance. For example, the parameter instances $\langle user1\rangle$ and $\langle user2\rangle$ are not compatible; the parameter instances $\langle user1\rangle$ and $\langle user1, order1\rangle$ are compatible, but their combination gives $\langle user1, order1\rangle$, same as one of the existing parametric instances.

At each reducer, we thus partition the parametric instances into groups based on their domains. In other words, θ_1 and θ_2 are in the same group if and only if $Dom(\theta_1) = Dom(\theta_2)$. Then we can safely skip the combinations of parametric instances in the same group, and the combinations of parametric instances between two groups where one group's domain subsumes another's.

Inverted index optimization. Since $\Delta_\perp$ is unchanged after being initialized, we build an inverted index for each parametric instance group in $\Delta_\perp$. For each value v_x of a parameter x, we record the set of parametric instances $\theta_\perp$ in this group such that $\theta_\perp(x) = v_x$. During the while loop at line 6 of the REDUCE function, the effort for finding strongly compatible parametric instances in $\Delta_\perp$ for a given parametric instance θ can be reduced using this inverted index. For each parametric instance group in $\Delta_\perp$, we first calculate the shared parameters with θ. For each shared parameter x, we lookup the index for $v_x = \theta(x)$ to get the set of parametric instances, each of which shares the same parameter value with θ on the parameter x. Thus, the set of strongly compatible instances for θ in this group is obtained by intersecting these sets of parametric instances for all shared parameters.

For example, suppose $\Delta_\perp$ contains the following parametric instances: $\langle user1\rangle$, $\langle user3\rangle$, $\langle user1, order1\rangle$ and $\langle user3, order1\rangle$. These parametric instances can be divided into two groups based on their domains, where one group has the domain of $\{userid\}$, and another group has the domain of $\{userid, orderid\}$. The inverted index built for the first group is:

"$user1$" : $\langle user1\rangle$

"$user3$" : $\langle user3\rangle$

[3] Detailed proof can be found in Appendix A.

And the inverted index for the second group is:

"$user1$" : $\langle user1, order1\rangle$

"$user3$" : $\langle user3, order1\rangle$

"$order1$" : $\langle user1, order1\rangle$, $\langle user3, order1\rangle$

Let $\theta_t = \langle user1, order1, item1\rangle$ be a parametric instance, now consider to find in $\Delta_\perp$ all compatible parametric instances to θ_t. Note that the only parameter shared by θ_t and the first group is *userid*, and its value in θ_t is $user1$. We thus need only to consider the list indexed by "$user1$", which gives the only parameter instance $\langle user1\rangle$. Thus the compatible parametric instance for θ_t in the first group is $\langle user1\rangle$. Moreover, note that the set of shared parameters between θ_t and the second group includes *userid* and *orderid*, and their values in θ_t are $user1$ and $order1$, respectively. We then retrieve in the second group the lists of parametric instances indexed by "$user1$" and "$order1$", respectively. Their intersection gives $\langle user1, order1\rangle$. Thus $\langle user1, order1\rangle$ is the compatible parametric instance for θ_t in the second group.

6. Distributed model synthesis with MapReduce

Once the complete PTA has been generated, as previously shown, many off-the-shelf model synthesis algorithms [1,2,11] can be applied to infer the system model. However, since these are centralized algorithms and the PTA can be a very large data structure, we propose a distributed model synthesis algorithm based on k-tail [1] with MapReduce to improve efficiency.

Many existing model synthesis algorithms can be viewed as variants of k-tail [1,2,4,11,14], which take as input a PTA, then repeatedly merge states of the PTA based on some criteria to get the final model. However, the complex criteria in these algorithms make them difficult to parallelize. Thus, we decide to parallelize the k-tail algorithm as the first step, leaving the parallelization of more complex yet accurate algorithms as a future work.

The most expensive operation here is to decide which states can be merged. Our idea is to distribute the most expensive operations to a number of mappers. With the intermediate results computed by the mappers, the model construction is comparatively simple, and is performed by a single reducer.

6.1. Data encoding

To realize the distributed model synthesis algorithm with MapReduce, the intermediate results should be in the form of key–value pairs. Since the "value" here is a state, we need a mechanism to set a key for each state. Moreover, as states with the same key are grouped together by MapReduce, the key should convey information about the merged states.

Before discussing how to encode states, we first introduce some notations of the behavior model [1]. A behavior model M is defined as a finite-state automaton $M = (\Sigma, S, s_0, \sigma, F)$, where:

- Σ is the input alphabet, which is also the set of base events (Definition 1).
- S is a finite, non-empty set of states.
- $s_0 \in S$ is an initial state.
- σ is the state-transition function: $\sigma : S \times \Sigma \to 2^S$.
- $F \subseteq S$ is the set of final states.

Let $\sigma^* : S \times \Sigma^* \to 2^S$ be the extended transition function, i.e., $\sigma^*(s, \epsilon) = \{s\}$ and $\sigma^*(s, e\omega) = \bigcup_{s' \in \sigma(s,e)} \sigma^*(s', \omega)$. Denote the input PTA model as M_{PTA}, and the target finite-state model as M_{FSM}.

Let k be a predefined integer. Let $\omega \in \Sigma^*$ be a word, i.e. a trace of base events. Let $\Sigma^{\leq k} = \Sigma^0 \cup \Sigma^1 \cdots \cup \Sigma^k$, then $\omega \in \Sigma^{\leq k}$ is a word of maximum length k, called a *k-word*. Given an automaton M, let f be a function from $S \times \Sigma^* \to \mathbb{B}$ such that for any state $s \in S$ and any word $\omega \in \Sigma^*$, $f(s, \omega) = T$ iff starting from s, the word ω is *accepted* by σ^*.[4]

Definition 12. Let s_1, s_2 be two states in M, we say s_1 and s_2 are *k-equivalent*, if for any k-word $\omega \in \Sigma^{\leq k}$, $f(s_1, \omega) = T$ iff $f(s_2, \omega) = T$.

The k-equivalence class that contains s is

$$[s] = \{t \in S \mid s \text{ and } t \text{ are } k\text{-equivalent}\}.$$

All states in a k-equivalent class accept the same set of k-words, and can be merged. A k-equivalent class in M_{PTA} corresponds to a state in M_{FSM}. The function f can be lifted to a equivalent class: $\forall \omega \in \Sigma^{\leq k}$, $f([s], \omega) = f(s', \omega)$, where s' can be any state in $[s]$.

[4] We do not require that a word ends in a final state, as in [15].

```
1: function Map(state)
2:     compute signature sig of state by Definition 13;
3:     Output(sig, state);

4: function Reduce(sig, states[])
5:     Create a new state s_sig in M_FSM w.r.t. sig;
6:     if ∃s ∈ states[].s is an initial state then
7:         set s_sig as a initial state in M_FSM;
8:     if ∃s ∈ states[].s is a final state then
9:         set s_sig as a final state in M_FSM;

10: function PostReduce
11:     for each transition (s_1, e, s_2) in M_PTA do
12:         add a transition ([s_1], e, [s_2]) in M_FSM;
```

Fig. 10. Distributed model synthesis.

Lemma 2. *For any two distinct k-equivalent classes $[s]$ and $[t]$, there must exist a k-word $\omega \in \Sigma^{\leq k}$, such that $f([s], \omega) \neq f([t], \omega)$.*

Proof. Assume the statement does not hold. Then $\forall s' \in [s]$, $\forall t' \in [t]$, and $\forall \omega \in \Sigma^{\leq k}$, $f(s', \omega) = f(t', \omega)$, which means s' and t' are k-equivalent. Thus $[s]$ and $[t]$ should be the same k-equivalent class, which is a contradiction. □

We can use the valuations of $f([s], \omega)$ for all $\omega \in \Sigma^{\leq k}$ to characterize $[s]$. Assume words in $\Sigma^{\leq k}$ to be indexed from 1 to $|\Sigma^{\leq k}|$. We use following definition to compute the signature of a state.

Definition 13. Let s be a state in S, the *signature* sig of s is a Boolean vector of length $|\Sigma^{\leq k}|$, such that $sig[i] = T$ iff with the i-th k-word ω in $\Sigma^{\leq k}$, $f(s, \omega) = T$ for $1 \leq i \leq |\Sigma^{\leq k}|$.

By Lemma 2, the signatures of s and t are identical, if and only if they are in the same k-equivalent class. We thus choose the signature of a given state as its key.

6.2. Mapper and reducer

The pseudocode of distributed model synthesis is shown in Fig. 10. Let S_i be the set of states distributed to $Mapper_i$. For each state $s \in S_i$, $Mapper_i$ computes the signature sig for s, and outputs the signature–state pair.

When all states signatures have been computed, the synthesis of M_{FSM} is simple, and can be performed by a single reducer. MapReduce sorts all signature–state pairs and puts the states with the same signature into one list. Let $states[]$ be the list of states with the same signature sig. The Reduce function is called for each pair of sig and $states[]$, and simply creates a new state s_{sig} in M_{FSM} for the given signature. If there exists a state $s \in states[]$ such that s is an initial state or a final state, then s_{sig} is set as an initial state or a final state in M_{FSM} correspondingly.

After all signatures have been processed, the PostReduce function is invoked, which adds transitions to M_{FSM}. For each transition in M_{PTA} from s to t due to the event e, a transition from $[s]$ to $[t]$ labeled e is added into M_{FSM}. The PostReduce function is called once and returns the synthesized model M_{FSM}.

We also implemented a Combine function to reduce the number of intermediate results. This function is called at each mapper, and simply merges the states corresponding to the same signature into a list. Since the Combine function is fairly simple, its pseudocode is omitted.

Note that the value of k can influence the performance of the k-tail algorithm. Intuitively, for larger k, it takes more time to compute each state signature from Definition 13. However, the impact of k on our distributed k-tail algorithm is often marginal since the state signatures are computed by mappers in parallel and k is often chosen as 2 or 3 in practice [4].

Correctness. Hereafter we discuss the correctness of our model synthesis algorithm, i.e., the model generated by our distributed algorithm is isomorphic to the one output by the original k-tail algorithm. Note that given an input PTA, the original k-tail algorithm always produce a same (non-deterministic) automaton as output [1]. Since we merge k-equivalent states in parallel, we only need to show that the order of state merging has no influence on the output model. In other words, merging two k-equivalent states in the input model does not influence the k-words accepted by all other states.

Given an automaton M, denote $word^k_M(s) = \{\omega | \omega \in \Sigma^{\leq k} \wedge f(s, \omega) = T\}$, i.e., the set of k-words accepted by the state s of M. Then we have the following theorem.[5]

Theorem 2. *Given an automaton M and two k-equivalent states s_1 and s_2, let M' be the automaton obtained by merging s_1 and s_2 into s'. The following statements hold:*

- $word^k_M(s_1) = word^k_M(s_2) = word^k_{M'}(s')$, *and*
- *for any state $s \in S' \setminus \{s'\}$, $word^k_M(s) = word^k_{M'}(s)$.*

[5] Detailed proof can be found in Appendix B.

```
1: model ← RandomModel(numState, maxTrans)
2: Γ ← {}
3: while sum(Γ) < threshold do
4:     t ←Simulate(minSteps, maxSteps)
5:     Γ ← Γ ∪ {t}
6: τ ←Interleave(Γ)
7: τ ←AddIrrelevant(τ)
```

Fig. 11. Pseudocode for synthesizing logs.

Intuitively, the first statement tells that the k-words accepted by s_1 and s_2 remain unchanged for the merged state s', while the second statement shows that the merging process has no influence on k-words accepted by other states. Thus, the order of state merging does not influence the output model, which directly implies the states can be merged in parallel and the model output by our approach is identical to the one generated by the original algorithm. However, states often accept more words longer than k during the state merging process such that the output model is a generalization of the original PTA.

7. Experimental evaluation

We implemented our approach on top of Hadoop 1.2.1,[6] and conducted experiments on Amazon Elastic MapReduce clusters.[7] Each computing node in the cluster has a dual-core CPU and 7.5 GB of memory. We configured each node to run two mappers and one reducer simultaneously. The running time spent on both MapReduce jobs (trace slicing and model synthesis) is measured separately. Each experiment has been performed 3 times, and the average value is reported.

We performed two groups of experiments to evaluate the performance of our approach. The first group of experiments are conducted on synthetic logs. This experiment group evaluates our approach under various settings. The second group of experiments are conducted on real logs from the DaCapo-9.12 suite [16]. This experiment group is to evaluate the practicality of our approach.

7.1. Experiments on synthetic logs

The pseudocode for synthesizing logs are given in Fig. 11. Firstly, an automaton is randomly generated (at line 1) as the target model to be inferred. This automaton is fixed to contain 50 states (i.e., $numState = 50$). The number of transitions exiting any state is between 1 to 5 (i.e., $maxTrans = 5$). Moreover, the model is generated to be connected, i.e., there exists a path between any two states. The parametric traces are randomly generated by simulations on the automaton (line 4). Each simulation starts from the initial state, and walks through the automaton by randomly taking the next transition. The minimal steps $minSteps$ and the maximal steps $maxSteps$ of each simulation are set to 10 and 100, respectively. As a result, the length of the generated trace is between 10 to 100. The generation of traces continues until the total length $sum(\Gamma)$ of all generated traces exceeds a predefined $threshold$ (line 3). At line 6, all generated parametric traces are randomly interleaved. At line 7, irrelevant entries are randomly added to the log as noise.

The default values for other parameters are: $|\mathcal{E}| = 15$, $|X| = 4$ and $k = 2$. Recall that $|\mathcal{E}|$ is the number of base events, $|X|$ the number of parameters, and k controls the merging criteria of the k-tail algorithm. The event definition function $\mathcal{D}_e$ is randomly determined, and its parameter values are randomly chosen from the integer domain. The size of the largest log file used in our experiments exceeds 10 GB.

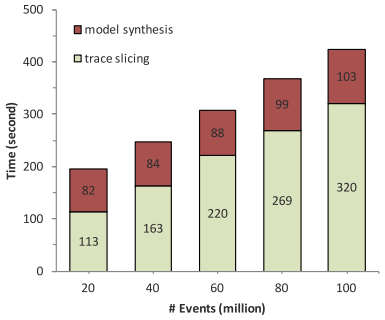
We performed several experiments to evaluate our approach under different settings, including basic performance, speed-up, scalability and the impact of some parameters. Note that our model synthesis algorithm is just a distributed implementation of k-tail. Given the same PTA, both algorithms infer the equal model (Theorem 2). In other words, our model synthesis algorithm has the same accuracy as the k-tail. Giving that the accuracy of the k-tail algorithm has been well studied in [17], we do not measure the accuracy of the inferred model in this paper.

Basic performance. The first experiment tests the running time of our approach for logs with increasing size, where size refers to the number of events in the log. All logs were generated by our random log synthesizer. Log size ranges from 20 to 100 million events. The computing cluster is fixed to have 10 machines.

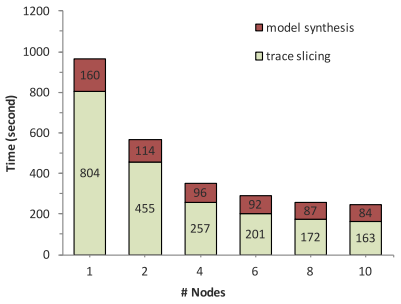
The experimental results are plotted in Fig. 12a. Each column in the graph contains two parts, representing the running time of trace slicing and model synthesis, respectively. The performance of our approach is very promising. The total processing time for the largest log (the file size exceeds 10 GB) is less than 7 minutes. One may observe that trace slicing needs more execution time than model synthesis. This is reasonable since trace slicing has to process the original log file and its complexity is inherently higher than model synthesis.

Speed-up. In the second experiment, we test the speed-up of our approach with increasing number of computing nodes. The log size is fixed to 40 million events, while the computing cluster's size varies from 1 node to 10 nodes.

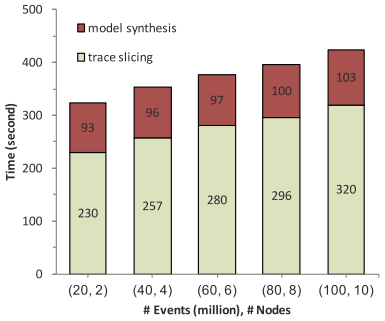
[6] http://hadoop.apache.org/.
[7] http://aws.amazon.com/elasticmapreduce/.



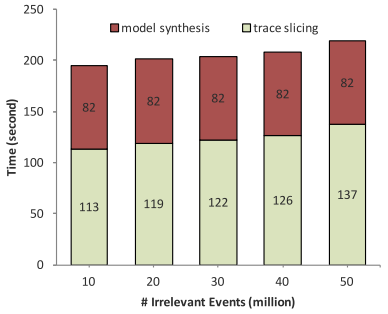
(a) Running time with increasing log size



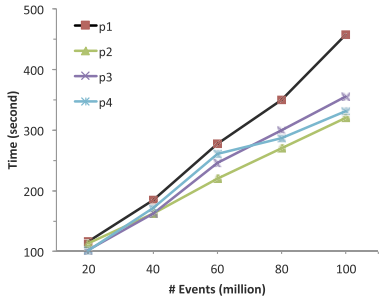
(b) Running time with increasing number of nodes



(c) Running time with increasing number of nodes and log size



(d) Running time with increasing number of irrelevant events



(e) Running time with varying $\mathcal{X}$

X \ $\mathcal{E}$	5	10	15
1	210/94	217/101	223/104
2	242/95	251/99	254/107
3	265/92	267/98	275/102
4	314/93	317/96	320/103

(f) Running time with varying $|\mathcal{E}|$ and $|X|$

Fig. 12. Experimental results on synthetic logs.

The experimental results are plotted in Fig. 12b. We observed that the total running time of our approach decreases considerably when given more computing nodes. This is well understandable. Moreover, along with the increase of computing nodes, the speed-up ratio goes down slowly. This is also reasonable, since the communication cost increases and there are some operations (for example, the Reduce and PostReduce functions in model synthesis) that cannot be parallelized or completely parallelized.

We also implemented a centralized version of the k-tail algorithm. This centralized algorithm takes 124 seconds to complete the model synthesis task in the same experiment. In contrast, our distributed k-tail algorithm needs 114 seconds or less for running on two or more computing nodes.

Scalability. The third experiment tests the scalability of our approach. We increase the log size (from 20 million to 100 million events) and the cluster size (from 2 to 10 nodes) by the same factor, and then observe the running time of our approach. Note that the ratio between log size and cluster size remains unchanged.

The experimental results are shown in Fig. 12c. When both log size and cluster size increase, the total running time increases a little. This phenomenon is very encouraging, which means our approach scales well.

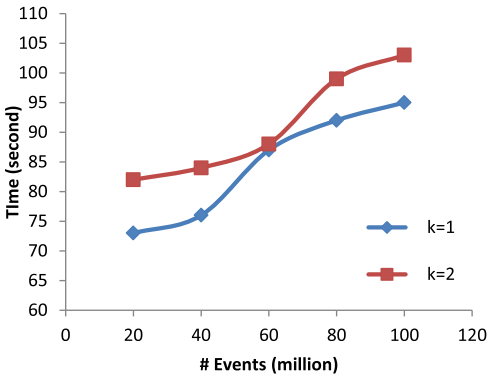


Fig. 13. Running time of model synthesis with varying k.

Impact of irrelevant events. The fourth experiment tests the impact of irrelevant events on the performance of our approach. The cluster size is fixed to 10 nodes. Among the tested logs, the number of relevant events is fixed to 10 million, while the number of irrelevant events varies from 10 million to 50 million.

The running time of our approach is shown in Fig. 12d. We can observe that the total running time almost does not change when the number of irrelevant events increases. This phenomenon explains that the irrelevant events have little impact on the running time of our approach. This feature is important since in real cases the percentage of irrelevant events may be very high.

Impact of the parameter window $\mathcal{X}$. This experiment is to evaluate the impact of $\mathcal{X}$ on the running time of trace slicing and the effectiveness of our heuristics for determining $\mathcal{X}$. In this experiment, X is fixed to contain four parameters p_1, p_2, p_3 and p_4; and $\mathcal{X}$ can be any singleton set of these four parameters. The parametric event definition is generated such that $\mathcal{X} = \{p_1\}$ corresponds to the biggest $\hat{T}_2$, and $\mathcal{X} = \{p_2\}$ corresponds to the smallest $\hat{T}_2$ (thus chosen by our heuristics in Section 5.1.1). The cluster size is fixed to 10 nodes, and the log size ranges from 20 million to 100 million events.

The running time is shown in Fig. 12e. As we can see, different settings of $\mathcal{X}$ have a significant impact on the trace slicing task: $\mathcal{X} = \{p_1\}$ yields the longest running time, while $\mathcal{X} = \{p_2\}$ yields the shortest running time.

Impact of $|\mathcal{E}|$ and $|X|$. The sixth experiment measures the performance of our approach under different settings of base events and parameters. We vary $|\mathcal{E}|$ from 5 to 15 and $|X|$ from 1 to 4. The log size is fixed to 100 million events and the cluster size is fixed to 10 nodes.

The running time for each setting of $|\mathcal{E}|$ and $|X|$ is shown in Fig. 12f, where each cell lists two numbers, reporting the running time for trace slicing and model synthesis, respectively. The trace slicing job is mainly influenced by $|X|$, and is much faster when X contains fewer parameters. While the model synthesis job is slightly impacted by $|\mathcal{E}|$, since the input PTA and the output model become more complex when $\mathcal{E}$ contains more base events. Also, our approach performs well in all settings of $|\mathcal{E}|$ and $|X|$.

Impact of k. The last experiment measures the impact of k (for k-tail) on our distributed model synthesis algorithm. We repeat the experiment on basic performance with different values of k. We only experimented with $k = 1, 2$ since larger k makes the synthesized model too complex and is usually not used in practice.

The results are summarized in Fig. 13. As the results show, the execution time of distributed model synthesis is higher for $k = 2$, which is obvious since the time of computing state signatures increases. A more interesting observation is that the increment of time for $k = 2$ is only about 8%. We believe this is mainly due to the parallel processing capacity of our distributed approach.

7.2. Experiments on real logs

To further evaluate the practicality of our approach, we carried out another group of experiments on real logs generated from the DaCapo-9.12 benchmark [18]. We monitored the execution of DaCapo benchmark programs to log all invocations of JDK 8 APIs (as events) using AspectJ.[8] The log size is about 10 GB and contains 85 million events. Our approach is then applied to this log to mine the behavior models of some Java classes from packages java.util, java.io and java.lang etc. For each Java class, its parametric event definition is defined on its set of public methods (method name as base event, method formal parameters as event parameters). The computing cluster contains 10 machines, and the parameter k for the k-tail algorithm is set to 2.

The experimental results are summarized in Fig. 14, where the columns from left to right give the Java class of interest, the number of events in the log, the execution time for trace slicing, and the execution time for model synthesis, respectively. The results show that our approach is able to handle real logs within reasonable amount of time. Most classes (all

[8] http://eclipse.org/aspectj/.

Java class	Events	Trace slicing (s)	Model synthesis (s)
java.util.HashMap	51,100	76	39
java.io.InputStream	591,380	77	52
java.io.OutputStream	26,301,320	213	89
java.io.Reader	1,253,840	86	69
java.io.File	148,060	83	70
java.lang.String	1,459,540	76	65
java.security.MessageDigest	55,482,840	254	59

Fig. 14. Experiment results on real logs.

but "OutputStream" and "MessageDigest") have only a few events (compared to the total number of 85 million events) in the log. Thanks to the parallel processing capacity of our MapReduce framework, the inference tasks for all classes can be finished in 2 to 5 minutes.

8. Discussion

In this section, we discuss two possible extensions of our approach, i.e., mining temporal invariants and log preprocessing.

8.1. Invariant mining

Our approach can be extended to support mining temporal invariants. Temporal invariants reflect the temporal relationship among events, and can improve the accuracy of the inferred model by preventing over-generalization [4,19]. For the shopping system example, one typical temporal invariant is that *create_order* must happen before *pay_order*.

In practice, the mined temporal invariants usually take some simple forms, such as $a \rightarrow b$, which means the occurrence of event a must eventually be followed by the event b. To mine these kinds of invariants, the occurrence information of events needs be recorded [19]. For example, let $occurrence[a]$ be the number of occurrences of event a, and $follows[a][b]$ the number of occurrences of event a that is followed by the event b, and $precedes[a][b]$ the number of event b that is preceded by the event a. Then with this information, we can easily decide if $a \rightarrow b$, $a \nrightarrow b$ or $a \leftarrow b$ holds or not.

The implementation is also simple. Each reducer needs to keep values for above three predicates for all base events. Each time a trace slice is constructed, the corresponding values are updated. When all reducers finish, these values should be merged and the temporal variants can be inferred based on the merged values.

8.2. Log preprocessor

In our approach, trace slicing and model synthesis are two separate MapReduce jobs. This design scheme gives users the flexibility to apply our technique as a log preprocessor.

Note that most of the existing model synthesis algorithms [1,2,4] share a similar work flow: they first construct a PTA as the initial model from the given traces, and then iteratively merge equivalent states in the PTA until the final model is inferred.

Thus, our approach can be easily combined with the existing model synthesis algorithms to infer more accurate models. The user needs to run the trace slicing job only. Based on the powerful data processing ability of MapReduce, the original log can be parsed, processed, and sliced efficiently. Then the generated PTA and optionally temporal constraints can be fed to the existing model synthesis algorithms to produce the system model.

9. Related works

A rich literature exists on inference of finite state models from a set of traces. k-tail [1] infers a finite state machine by iteratively merging k-equivalent states, which can be seen as a basis of many existing algorithms. sk-strings [2] infers probabilistic finite state machine by merging states which are indistinguishable by their top s percent of the most probable k-strings. RNPI [20] considers both positive and negative samples, and infers a model which accepts all positive samples but none negative samples. Evidence-driven state merging (EDSM) [21], e.g., Blue-Fringe [21], ranks mergeable states with statistical information to determine which pair should be merged first. The work [15] considers three algorithms to infer process models from event data, including neural network, k-tail, and Markov methods. SMArTIC [22] improves the accuracy of the inferred model by first clustering relevant traces together and merging the models inferred from each cluster. kBehavior [23] infers models incrementally using heuristics based on recurrent patterns in the trace. Li et al. [24] propose a model inference technique for digital circuits by first mining recurring patterns from traces using predefined templates and then synthesizing them into complex models. Heule and Verwer [25,26] propose an efficient heuristics for EDSM using SAT solvers by encoding the model inference problem into Boolean formulas. InvariMint [27] is a general framework for declaratively specifying model inference algorithms. In InvariMint, a model inference algorithm is expressed as a set of properties expected to hold on the inferred model. To address the scalability issue of model inference, Busany and Maoz [28] take a

statistical approach by inferring a model from a sample of traces with certain statistical guarantees. The basic idea is to consider model inference as a statistical experiment, where each trace in the log is treated as a trial to either accept or reject the inferred model. However, these works do not take into consideration event parameters, which are important for software systems, and are all centralized algorithms.

Although many existing works, including ours, infer models passively from a set of given traces, the inference process can also be active. Angluin [29] proposes the first active model inference algorithm L^*, which assumes an oracle to answer two kinds of queries, i.e., membership queries to determine whether a trace is accepted, and equivalence queries to check the correctness of the hypothesized model. QSM [30] employs a query-drive state merging strategy, and submits membership queries of generalized traces to the end user during the state merging process. However, QSM is often infeasible in practice as it may submit too many queries for even a simple model. To alleviate this, Walkinshaw et al. [31] use a testing framework to answer membership queries. Howar [32] extends L^* [29] to support alphabet refinement. It abstracts a set of concrete events with parameters into abstract events, which are refined subsequently to remove non-determinism caused by abstraction.

Incorporating temporal constraints can improve the accuracy of the inferred model by preventing over-generalization. Walkinshaw and Bogdanov [33] take LTL formulas as additional inputs, and use a model checker to iteratively refine the inferred model until all LTL constraints are satisfied. Lo et al. [4] automatically discover temporal constraints from traces with predefined templates, and only merge two states without violating any temporal constrains. This work also proposes an approximate merging criteria to avoid the high complexity incurred by checking temporal constraints. Lamprier et al. [14] improve the previous work [4] and propose an exact but efficient state merging strategy by maintaining an auxiliary structure for each state. Synoptic [19] adopts a similar approach, but employs partition-based abstraction to infer an initial model and iteratively refines the model using the counterexample-guided abstraction refinement approach [34]. SpecForge [35] combines multiple model inference algorithms. It first decomposes models inferred by existing algorithms into temporal constraints, filters outliers, and converts the constraints into the final model. As mentioned in Section 8, our approach can be easily extended to support mining temporal constraints, and it is an interesting future direction to integrate temporal constraints into our model synthesis phase.

Ammons et al. [3] apply model inference techniques to infer software API specifications. They first split program traces into interaction scenarios based on data dependence information, and infer a specification model using an existing learner [2]. Mariani and Pastore [36] infer behavior models from log files to locate system failures by first replacing concrete event parameters with symbolic ones using heuristics. The difference is that the previous works either require the user manually identify related parameters [3] or preprocess traces using heuristics [36], while our approach performs trace slicing based on parameter instances with little human effort. Lee et al. [8] propose a trace slicing technique to handle parametric traces, which also inspired our work. However, their trace slicing algorithm is centralized, and seems to be a bottleneck of the model inference process.

Instead of inferring behavior models only from execution traces, other works take program source code into consideration. Whaley et al. [37] combine static and dynamic analysis to infer component inference models. They first groups public methods of a component based on the accessed fields, and infers a model for each group representing admissible method sequences. The inferred models are further enhanced using dynamic analysis with execution traces. JIST [38] abstracts a concrete class using predicate abstraction, and infers a specification model by solving a two player game between the abstract class and the safety requirement using the L^* algorithm [29]. Wasylkowski et al. [39] introduce a static method to infer object usage models from client code by first abstracting source code into a method model, which is projected onto objects to infer object usage models. Shoham et al. [40] mine API specifications from client code with two steps by first collecting all possible event traces using abstract interpretation [41], and then summarizing the collected traces to remove noises and infer specifications. SEIM [42] uses inter-procedural analysis to infer interaction models from client code, and presents a refinement strategy to eliminate infeasible behaviors. de Caso et al. [43] infer behavior models from method pre/post-conditions to support contract validation using enabledness-preserving abstraction. Krka et al. [44] considers two strategies to combine model inference with program invariants, namely state-enhanced k-tail (SEKT) and trace-enhanced MTS inference (TEMI). SEKT only merges state pairs having identical internal states, while TEMI refines the modal transition system (MTS) constructed from program invariants based on execution traces. Our work differs from these works in that we treat the system as a black-box, and only require the execution logs instead of the source code.

However, it has been suggested that finite state models sometimes are insufficient for software systems as information of data values is lacking [45–47]. In general, there are two directions to extend finite state models with data values. The first is to enrich finite state models with data states. Obstra [48] abstracts concrete objects from test cases to infer object states models, which are further augmented with additional tests to discover more complete behavior. ADABU [45] distinguishes mutator methods and inspector methods of an object. It calls inspector methods after each mutator method to inspect object states, and maps these concrete states to abstract states to infer the object model. Walkinshaw et al. [49] discover state transitions from programs using symbolic execution techniques, but require the user to manually define data states. SPY [50] generalizes the partial model inferred execution traces using graph transformation rules to obtain behavior models. TAUTOKO [51] enriches existing model inference algorithms with test case generation to discover more comprehensive behavior.

Another direction is to infer extended finite state machines (EFSM), where transitions are labeled with guard predicates. Berg et al. [52] adapt the L^* algorithms [29] to infer parameterized models, but only boolean predicates are allowed. GK-

tail [46] extends k-tail [1] to infer EFSMs from execution traces. It first merges traces with same events ignoring data values, and generates predicates for merged traces using Daikon [53]. Finally, the EFSM is inferred by merging equivalent states as in k-tail [1]. Psyco [54] infers component interface models using the L^* algorithm [29], where events are augmented with predicates, and uses symbolic execution to answer queries. It also performs alphabet refinement to remove non-determinism. X-Psyco [55] improves Psyco [54] in several ways. X-Psyco uses partial order reduction to reduce the generated method sequences, and relies on concrete executions for model inference, which is combined with symbolic execution to ensure completeness and generate transition guards. Tzuyu [47] adopts a similar framework, but employs system testing as the oracle and uses SVM [56] to perform alphabet refinement. MINT [57] is a general approach for inferring EFSMs, which first uses data classifiers to infer guard predicates, and iteratively merge compatible states to obtain the final model. Comparing with these works, our approach only uses data values to perform trace slicing, while the values are discarded in the inferred model. However, it is worth considering the parallelization of these model inference algorithms.

Many inference algorithms of other finite state models have also been proposed recently, such as mealy machines [58–60] which model output values, register automata [61–63] which support variables, hybrid automata [64] modeling continuous system behavior [64], discrete time Markov chains [5] to model user navigational behavior, communicating finite state machines [6] to support concurrent systems, and resource finite state machines [65] to express resource usages etc. However, to the best of our knowledge, there is no previously published work on applying MapReduce to model inference.

Recently, Wang et al. [9] studied the parallelization of specification mining using MapReduce. Their parallelization solution for the k-tail algorithm is to divide the original trace into several groups, on each of which an instance of the original k-tail algorithm is then executed. Our parallelization solution is at a finer-grained level in the sense that our approach parallelizes the inference of a single behavior model.

Moreover, since our approach involves log processing, our work also shares some similarities with trace checking with MapReduce. Informally, trace checking is the problem to check the compliance of a log of traces with temporal formulas. However, since logs can be potentially very large, several MapReduce-based trace checking algorithms have recently been proposed. Barre et al. [66] present an algorithm for checking Linear Temporal Logic (LTL) formulas over event traces with MapReduce. The algorithm checks the formula by iteratively processing the logical operators in the same level in a bottom-up manner. Thus, the parallelism is bounded by the structure of the input formulas. Bianculli et al. [67] improve the previous work [66] by supporting Metric Temporal Logic (MTL) [68], where temporal operators are enhanced with time intervals, with aggregating modalities. Bersani et al. [69] further propose a novel lazy semantics for MTL to handle the memory scalability issue caused by the bounds of time intervals. The lazy semantics allows to decompose a MTL formula with large time intervals into an equivalent formula with smaller bounds. Basin et al. [70] present a formal log slicing framework for checking Metric First-Order Temporal Logic (MFOTL) [71] policies, which is further implemented with MapReduce. The basic idea to slice a log into several pieces, check each piece separately, and merge the results from all these pieces together. But the major difference between our approach and these works is that we mainly focus on behavior model inference from large logs, rather than checking compliance with temporal logics.

10. Conclusion

In this paper, we presented an approach to infer software behavior models from large logs using MapReduce. In our approach, the logs are first parsed and sliced, then the model is inferred by the distributed k-tail algorithm. Our approach can also be used as a log preprocessor and combined with existing model inference algorithms. Experiments on Amazon clusters and large datasets show the efficiency and scalability of our approach. This paper extends our previous work [12] in several ways. Specially, we describe several practical optimizations, formally prove the correctness of our approach, and provide more complete experimental assessment under various settings.

We plan to perform case studies on large logs generated by real software systems to further evaluate the performance and applicability of our approach. We also plan to investigate the parallelization of more accurate model inference algorithms or incorporating temporal constraints during the inference phase.

Acknowledgements

We thank Dr. Srdjan Krstic for his comments to our manuscript. This work was supported in part by the Chinese National 973 Plan (2010CB328003), the NSF of China (61672310, 61272001, 91218302) and the Chinese National Key Technology R&D Program (SQ2012BAJY4052).

Appendix A. Correctness of distributed trace slicing

In the following, we show the correctness of our distributed trace slicing algorithm. First, we show some properties of the connected parametric instance (Definition 9).

Lemma 3. *Given a parametric trace τ and a τ-connected parametric instance θ, there exists a sequence of parametric events $e_1\langle\theta_1\rangle, \ldots, e_n\langle\theta_n\rangle$ such that:*

- $\theta = \theta_1 \sqcup \ldots \sqcup \theta_n$, *and*
- *for any* $e_i\langle\theta_i\rangle$ *and* $e_j\langle\theta_j\rangle$ *in the sequence,* θ_j *is* reachable *from* θ_i*, i.e., there exist* $e'_1\langle\theta'_1\rangle, \ldots, e'_k\langle\theta'_k\rangle$ *in the sequence such that* $\theta_i \bowtie \theta'_1, \ldots, \theta'_{k-1} \bowtie \theta'_k, \theta'_k \bowtie \theta_j$.

Proof. We show the lemma with structural induction over the definition of connectedness (Definition 9).

Inductive basis: if $e\langle\theta\rangle \in \tau$, let the sequence be $e\langle\theta\rangle$, i.e., with only one parametric event. Apparently it satisfies above conditions. Thus the lemma holds.

Inductive step: suppose there exist θ_1 and θ_2 such that both θ_1 and θ_2 are τ-connected, $\theta_1 \bowtie \theta_2$ and $\theta = \theta_1 \sqcup \theta_2$. From the inductive assumption, the lemma holds for both θ_1 and θ_2. Let $\theta_1 = \theta_1^1 \sqcup \ldots \sqcup \theta_1^u$ and $\theta_2 = \theta_2^1 \sqcup \ldots \sqcup \theta_2^v$, where $e_1^1\langle\theta_1^1\rangle, \ldots e_1^u\langle\theta_1^u\rangle, e_2^1\langle\theta_2^1\rangle, \ldots, e_2^v\langle\theta_2^v\rangle \in \tau$. Then $\theta = \theta_1 \sqcup \theta_2 = (\theta_1^1 \sqcup \ldots \sqcup \theta_1^u) \sqcup (\theta_2^1 \sqcup \ldots \sqcup \theta_2^v)$. From the definition of the combination operator (Definition 7), it is straightforward that $\sqcup$ is associative. Thus, $\theta = \theta_1^1 \sqcup \ldots \sqcup \theta_1^u \sqcup \theta_2^1 \sqcup \ldots \sqcup \theta_2^v$, and the first statement holds.

For the second statement, let $e_i\langle\theta_i\rangle$ and $e_j\langle\theta_j\rangle$ be two parametric events in the sequence of $e_1^1\langle\theta_1^1\rangle, \ldots e_1^u\langle\theta_1^u\rangle, e_2^1\langle\theta_2^1\rangle, \ldots, e_2^v\langle\theta_2^v\rangle$. From the inductive assumption, we only need to consider the case where $e_i\langle\theta_i\rangle$ in $e_1^1\langle\theta_1^1\rangle, \ldots e_1^u\langle\theta_1^u\rangle$ and $e_j\langle\theta_j\rangle$ in $e_2^1\langle\theta_2^1\rangle, \ldots, e_2^v\langle\theta_2^v\rangle$, since otherwise the statement trivially holds from the inductive assumption. Since $\theta_1 \bowtie \theta_2$, there must exist a parameter $x \in X$ such that both $\theta_1(x)$ and $\theta_2(x)$ are defined. Further from $\theta_1 = \theta_1^1 \sqcup \ldots \sqcup \theta_1^u$ and $\theta_2 = \theta_2^1 \sqcup \ldots \sqcup \theta_2^v$, there must exist parametric instances θ_1^i and θ_2^j such that both $\theta_1^i(x)$ and $\theta_2^j(x)$ are defined. Then, since θ_1 is compatible with θ_2, and $\theta_1^i \sqsubseteq \theta_1$ and $\theta_2^j \sqsubseteq \theta_2$, we have θ_1^i is compatible with θ_2^j. Thus, $\theta_1^i \bowtie \theta_2^j$. Moreover, from the inductive assumption, we have θ_1^i is reachable from θ_i, and θ_j is reachable from θ_2^j. By concatenating two sequences with $\theta_1^i \bowtie \theta_2^j$, we have θ_j is reachable from θ_i, and the second statement holds. □

Lemma 4. *Given a parametric trace* τ *and a parameter window* $\mathcal{X}$*, for any complete and connected parametric instance* θ*, there exists a parametric event* $e'\langle\theta'\rangle \in \tau$ *such that* $\theta' \upharpoonright_{\mathcal{X}} = \theta \upharpoonright_{\mathcal{X}}$.

Proof. From Lemma 3, since θ is connected, there exists a sequence of parametric events $e'_1\langle\theta'_1\rangle, \ldots e'_n\langle\theta'_n\rangle \in \tau$ such that $\theta = \theta'_1 \sqcup \ldots \sqcup \theta'_n$. Moreover, from the definition of $\mathcal{X}$ (Definition 10), for any $e'_i\langle\theta'_i\rangle$ in the sequence, either $Dom(\theta'_i) \cap \mathcal{X} = \emptyset$ or $\mathcal{X} \subseteq Dom(\theta'_i)$. Now suppose for any θ'_i, $Dom(\theta'_i) \cap \mathcal{X} = \emptyset$. Then, we have $Dom(\theta'_1 \sqcup \ldots \sqcup \theta'_n) \cap \mathcal{X} = \emptyset$, which means θ is not complete and leads to a contradiction. Thus, there must exist some $e'\langle\theta'\rangle$ in the sequence such that $\mathcal{X} \subseteq Dom(\theta')$. Then from the definition of the combination operator $\sqcup$ (Definition 7), we have $\theta' \upharpoonright_{\mathcal{X}} = \theta \upharpoonright_{\mathcal{X}}$. □

Now, we consider the while loop at line 6 of the REDUCE function, which leads to the following lemma.

Lemma 5. *Given a parametric trace* τ *and a complete and connected parametric instance* θ*, for any parametric event* $e'\langle\theta'\rangle \in \tau$ *such that* $\theta' \sqsubseteq \theta$, $\Delta(\theta')$ *is defined after the while loop at line 6 of the* REDUCE *function for* $key = \theta \upharpoonright_{\mathcal{X}}$.

Proof. Let *value*[] be the list of parametric events for $key = \theta \upharpoonright_{\mathcal{X}}$. If $\mathcal{X} \subseteq Dom(\theta')$, which means $key(e'\langle\theta'\rangle) = \theta' \upharpoonright_{\mathcal{X}} = \theta \upharpoonright_{\mathcal{X}}$, then $e'\langle\theta'\rangle \in values[]$. Thus, $\Delta(\theta')$ is defined after line 5 of the REDUCE function for $key = \theta \upharpoonright_{\mathcal{X}}$, and the statement holds.

Otherwise, $\mathcal{X} \cap Dom(\theta') = \emptyset$, which means $key(e'\langle\theta'\rangle) = \perp$. Thus, $\Delta_\perp(\theta')$ is defined after line 3 of the REDUCE function for $key = \perp$. From Lemma 3, since θ is connected, there exists a sequence of parametric events $e_1\langle\theta_1\rangle \ldots e_n\langle\theta_n\rangle \in \tau$ such that $\theta = \theta_1 \sqcup \ldots \sqcup \theta_n$. Then from the proof of Lemma 4 and without loss of generality, let $e_1\langle\theta_1\rangle$ in the sequence be the parametric event such that $\theta \upharpoonright_{\mathcal{X}} = \theta_1 \upharpoonright_{\mathcal{X}}$. From the previous discussion, $\Delta(\theta_1)$ is defined after line 5 of the REDUCE function for $key = \theta \upharpoonright_{\mathcal{X}}$.

Now consider θ'. Let $x \in X$ be a parameter such that $\theta'(x)$ is defined. Since θ is complete, i.e., $Dom(\theta) = Dom(\theta_1 \sqcup \ldots \sqcup \theta_n) = X$, there must exist a parametric event $e_i\langle\theta_i\rangle$ in the sequence $e_1\langle\theta_1\rangle \ldots e_n\langle\theta_n\rangle$ such that $\theta_i(x)$ is defined. Moreover, since both $\theta' \sqsubseteq \theta$ and $\theta_i \sqsubseteq \theta$, we have θ' is compatible with θ_i. Thus, $\theta' \bowtie \theta_i$. Then from Lemma 3, there exist $e'_1\langle\theta'_1\rangle, \ldots, e'_k\langle\theta'_k\rangle$ in the sequence of $e_1\langle\theta_1\rangle \ldots e_n\langle\theta_n\rangle$ such that $\theta_1 \bowtie \theta'_1, \ldots, \theta'_{k-1} \bowtie \theta'_k$ and $\theta'_k \bowtie \theta_i$. By concatenating with $\theta_i \bowtie \theta'$, we have θ' is reachable from $\bar{\theta}_1$. Note that for any $\bar{\theta}$ ($\bar{\theta} \sqsubseteq \theta$) in the sequence of $\theta'_1, \ldots \theta'_k, \theta_i$, either $\Delta(\bar{\theta})$ is defined after line 5 of the REDUCE function for $key = \theta \upharpoonright_{\mathcal{X}}$ ($\mathcal{X} \subseteq Dom(\bar{\theta})$), or $\Delta_\perp(\bar{\theta})$ is defined after line 3 of the REDUCE function for $key = \perp$ ($\mathcal{X} \cap Dom(\bar{\theta}) = \emptyset$). Thus, during the while loop at line 6 of the REDUCE function for $key = \theta \upharpoonright_{\mathcal{X}}$, $\Delta_\perp(\theta')$ is added into Δ by following the sequence of $\theta_1, \theta'_1, \ldots \theta'_k, \theta_i, \theta'$, and the statement also holds. □

Then, we show the trace slice $\tau \upharpoonright_\theta$ is correctly constructed at line 24, which is achieved by the following lemma.

Lemma 6. $\tau \upharpoonright_\theta$ *computed at line 24 satisfies the definition of* θ*-trace slice of* τ *(Definition 6).*

Proof. From the definition of trace slice (Definition 6), given a parametric instance θ and a parametric trace τ, θ-trace slice of τ consists of the base events of all parametric events $e'\langle\theta'\rangle \in \tau$ such that $\theta' \sqsubseteq \theta$. According to Lemma 5, for any $e'\langle\theta'\rangle \in \tau$ such that $\theta' \sqsubseteq \theta$, $\Delta(\theta')$ is defined at line 23 and $\Delta(\theta')$ is a sequence of base events for θ'. Moreover, it is trivial to see that

$\Delta(\theta')$ is defined only if $\theta' \in \tau$. Thus, $\tau \restriction_\theta$ can be computed by merging the lists of base events $\Delta(\theta')$ for $\theta' \sqsubseteq \theta$ in ascending order of *timestamp* at line 24. □

Finally, the following theorem states the correctness of our distributed trace slicing algorithm.

Theorem 1. *Given a parametric trace τ, the following statements hold for the distributed trace slicing job:*

1. *for any $\tau \restriction_\theta$ constructed at line 24, the parametric instance θ is complete and connected, and*
2. *for any complete and connected parametric instance θ, θ-trace slice of τ is constructed at line 24.*

Proof. The first statement trivially holds since the guards at lines 19 and 22 ensure only $\tau \restriction_\theta$ for complete and connected parametric instances are constructed.

We then mainly focus on the second statement. From Lemma 4, for any complete and connected parametric instance θ, there exists a parametric event $e'\langle\theta'\rangle \in \tau$ such that $\theta' \restriction_{\mathcal{X}} = \theta \restriction_{\mathcal{X}}$, which means there exists a key–value pair output by mappers with $key = \theta \restriction_{\mathcal{X}}$. Then we only need to show θ-trace slice of τ can be constructed in the REDUCE function for $key = \theta \restriction_{\mathcal{X}}$, which is equivalent to show $\theta \in \Omega$ at line 22 from Lemma 6. From Lemma 3, since θ is connected, there exists a sequence of parametric events $e_1\langle\theta_1\rangle, \ldots e_n\langle\theta_n\rangle \in \tau$ such that $\theta = \theta_1 \sqcup \ldots \sqcup \theta_n$, and for any $e_i\langle\theta_i\rangle$ and $e_j\langle\theta_j\rangle$ in the sequence, θ_j is reachable from θ_i. Moreover, from Lemma 5, all $\Delta(\theta_1), \ldots, \Delta(\theta_n)$ are defined after the while loop at line 6, i.e., $\theta_1, \ldots, \theta_n \in \Omega$ after line 18. Thus, θ is constructed during the while loop at line 19 by combining the parametric instances $\theta_1, \ldots \theta_n$, and the second statement holds. □

Appendix B. Correctness of distributed model synthesis

In this appendix we show the correctness of our distributed model synthesis algorithm. Recall that given an automaton M, denote $word^k_M(s) = \{\omega | \omega \in \Sigma^{\leq k} \wedge f(s, \omega) = T\}$, i.e., the set of k-words accepted by the state s of M. We further lift the above notation to a set of states S, i.e., $word^k_M(S) = \bigcup_{s \in S} word^k_M(s)$. Given a set of words Ω and a base event $e \in \Sigma$, denote $e \cdot \Omega = \{e \cdot \omega | \omega \in \Omega\}$, i.e., the set of words obtained by concatenating e with each word in Ω.

Now consider the state merging process. Given an automaton M and two k-equivalent states s_1 and s_2 in M, let M' be the automaton obtained by merging s_1 and s_2 into s'. Let S and σ be the set of states and the transition function of M, and S' and σ' be the set of states and the transition function of M'. It is straightforward that $S' = (S \setminus \{s_1, s_2\}) \cup \{s'\}$. The transition function σ' is obtained as follows. For the state s' and any $e \in \Sigma$, we have:

- $\sigma'(s', e) = \sigma(s_1, e) \cup \sigma(s_2, e)$, if $s_1, s_2 \notin \sigma(s_1, e) \cup \sigma(s_2, e)$;
- $\sigma'(s', e) = ((\sigma(s_1, e) \cup \sigma(s_2, e)) \setminus \{s_1, s_2\}) \cup \{s'\}$, if s_1 or s_2 in $\sigma(s_1, e) \cup \sigma(s_2, e)$.

For any state $s \in S' \setminus \{s'\}$ and any $e \in \Sigma$, we have:

- $\sigma'(s, e) = \sigma(s, e)$, if $s_1, s_2 \notin \sigma(s, e)$;
- $\sigma'(s, e) = (\sigma(s, e) \setminus \{s_1, s_2\}) \cup \{s'\}$, if $s_1 \in \sigma(s, e)$ or $s_2 \in \sigma(s, e)$.

We first show two properties of k-equivalent states.

Lemma 7. *Let s_1 and s_2 be two k-equivalent states, then for any $i \in [1, k]$, s_1 and s_2 are i-equivalent.*

Proof. Prove by contradiction. Assume s_1 and s_2 are not i-equivalent for some $i \in [1, k]$, which means there exists a word $\omega \in \Sigma^{\leq i}$ such that ω is accepted by only one of s_1 and s_2. Since $\Sigma^{\leq i} \subseteq \Sigma^{\leq k}$, we have $\omega \in \Sigma^{\leq k}$. Thus, s_1 and s_2 are not k-equivalent, which is a contradiction. □

Lemma 8. *Given two states s_1 and s_2 in an automaton M, s_1 and s_2 are k-equivalent iff for any base event $e \in \Sigma$, $word^{k-1}_M(\sigma(s_1, e)) = word^{k-1}_M(\sigma(s_2, e))$.*

Proof. $\Rightarrow$: Suppose s_1 and s_2 are k-equivalent. Let e be a base event, and $e \cdot \omega \in \Sigma^{\leq k}$ be a k-word starting from e, where $\omega \in \Sigma^{\leq k-1}$. Since s_1 and s_2 are k-equivalent, we have $word^k_M(s_1) = word^k_M(s_2)$, which implies $e \cdot \omega \in \bigcup_{e' \in \Sigma} e' \cdot word^{k-1}_M(\sigma(s_1, e'))$ iff $e \cdot \omega \in \bigcup_{e' \in \Sigma} e' \cdot word^{k-1}_M(\sigma(s_2, e'))$. Moreover, since $e \cdot \omega \notin \bigcup_{e' \in (\Sigma \setminus \{e\})} e' \cdot word^{k-1}_M(\sigma(s_1, e'))$ and $e \cdot \omega \notin \bigcup_{e \in (\Sigma \setminus \{e\})} e' \cdot word^{k-1}_M(\sigma(s_2, e'))$, we have $e \cdot \omega \in e \cdot word^{k-1}_M(\sigma(s_1, e))$ iff $e \cdot \omega \in e \cdot word^{k-1}_M(\sigma(s_2, e))$, which means $\omega \in word^{k-1}_M(\sigma(s_1, e))$ iff $\omega \in word^{k-1}_M(\sigma(s_2, e))$. Thus, the statement holds.

$\Leftarrow$: Let $e \in \Sigma$ be a base event and $word^{k-1}_M(\sigma(s_1, e)) = word^{k-1}_M(\sigma(s_2, e))$, which implies $e \cdot word^{k-1}_M(\sigma(s_1, e)) = e \cdot word^{k-1}_M(\sigma(s_2, e))$. Thus, $\bigcup_{e \in \Sigma} e \cdot word^{k-1}_M(\sigma(s_1, e)) = \bigcup_{e \in \Sigma} e \cdot word^{k-1}_M(\sigma(s_2, e))$, which implies $word^k_M(s_1) = word^k_M(s_2)$, and the statement holds. □

The following theorem ensures that the order of state merging has no influence on the output model.

Theorem 2. *Given an automaton M and two k-equivalent states s_1 and s_2, let M' be the automaton obtained by merging s_1 and s_2 into s'. The following statements hold:*

- $word^k_M(s_1) = word^k_M(s_2) = word^k_{M'}(s')$, *and*
- *for any state* $s \in S' \setminus \{s'\}$, $word^k_M(s) = word^k_{M'}(s)$.

Proof. Let S and σ be the set of states and the transition function of M, and S' and σ' be the set of states and the transition function of M'. Note that for the first statement, it is trivial that $word^k_M(s_1) = word^k_M(s_2)$, and we only need to consider $word^k_M(s_1) = word^k_{M'}(s')$. We then show the theorem by natural induction over the length l ($l \leq k$) of the words.

Inductive basis: $l = 1$. Let $e \in \Sigma$ be a base event, i.e., a word with the length 1. For the first statement, if $s_1, s_2 \notin \sigma(s_1, e) \cup \sigma(s_2, e)$, then $\sigma'(s', e) = \sigma(s_1, e) \cup \sigma(s_2, e)$. From Lemma 7, s_1 and s_2 are also 1-equivalent, which means $\sigma(s_1, e) \neq \emptyset$ iff $\sigma(s_2, e) \neq \emptyset$. Thus, we have $\sigma(s_1, e) \neq \emptyset$ iff $\sigma'(s', e) \neq \emptyset$. Otherwise, if s_1 or s_2 in $\sigma(s_1, e) \cup \sigma(s_2, e)$, then $\sigma'(s', e) = ((\sigma(s_1, e) \cup \sigma(s_2, e)) \setminus \{s_1, s_2\}) \cup \{s'\}$. In this case, both $\sigma(s_1, e) \neq \emptyset$ and $\sigma'(s', e) \neq \emptyset$, which implies $\sigma(s_1, e) \neq \emptyset$ iff $\sigma'(s', e) \neq \emptyset$. Thus, $e \in word^1_M(s_1)$ iff $e \in word^1_{M'}(s')$, and the first statement holds.

For the second statement, if $s_1, s_2 \notin \sigma(s, e)$, then $\sigma'(s, e) = \sigma(s, e)$. Thus, $\sigma(s, e) \neq \emptyset$ iff $\sigma'(s, e) \neq \emptyset$. Otherwise, if s_1 or $s_2 \in \sigma(s, e)$, then $\sigma'(s, e) = (\sigma(s, e) \setminus \{s_1, s_2\}) \cup \{s'\}$. In this case, both $\sigma(s, e) \neq \emptyset$ and $\sigma'(s, e) \neq \emptyset$, which also implies $\sigma(s, e) \neq \emptyset$ iff $\sigma'(s, e) \neq \emptyset$. Thus, $e \in word^1_M(s)$ iff $e \in word^1_{M'}(s)$, and the second statement holds.

Inductive step: assuming the theorem holds for l ($l \leq k - 1$), we show that the theorem holds for $l + 1$. Let $e \in \Sigma$ be a base event.

For the first statement, from Lemma 8, we only need to show $word^l_M(\sigma(s_1, e)) = word^l_{M'}(\sigma'(s', e))$. If $s_1, s_2 \notin \sigma(s_1, e) \cup \sigma(s_2, e)$, then $\sigma'(s', e) = \sigma(s_1, e) \cup \sigma(s_2, e)$. From the inductive assumption (the second statement), we have $word^l_M(\sigma(s_1, e) \cup \sigma(s_2, e)) = word^l_{M'}(\sigma(s_1, e) \cup \sigma(s_2, e))$. Moreover, from Lemma 7, we have s_1 and s_2 are $(l+1)$-equivalent, which further implies $word^l_M(\sigma(s_1, e)) = word^l_M(\sigma(s_2, e))$ from Lemma 8. Then, we have the following equation:

$$
\begin{aligned}
&word^l_{M'}(\sigma'(s', e)) \\
&= word^l_{M'}(\sigma(s_1, e) \cup \sigma(s_2, e)) \\
&= word^l_M(\sigma(s_1, e)) \cup word^l_M(\sigma(s_2, e)) \\
&= word^l_M(\sigma(s_1, e))
\end{aligned}
$$

Thus, the first statement holds in this case.

Otherwise, if s_1 or s_2 in $\sigma(s_1, e) \cup \sigma(s_2, e)$, then $\sigma'(s', e) = ((\sigma(s_1, e) \cup \sigma(s_2, e)) \setminus \{s_1, s_2\}) \cup \{s'\}$. From the inductive assumption (the second statement), we have $word^l_M((\sigma(s_1, e) \cup \sigma(s_2, e)) \setminus \{s_1, s_2\}) = word^l_{M'}((\sigma(s_1, e) \cup \sigma(s_2, e)) \setminus \{s_1, s_2\})$. Also from the inductive assumption (the first statement), we have $word^l_M(s_1) = word^l_M(s_2) = word^l_{M'}(s')$. Moreover, from the previous discussion, we have $word^l_M(\sigma(s_1, e)) = word^l_M(\sigma(s_2, e))$. Then, we have the following equation:

$$
\begin{aligned}
&word^l_{M'}(\sigma'(s', e)) \\
&= word^l_{M'}(((\sigma(s_1, e) \cup \sigma(s_2, e)) \setminus \{s_1, s_2\}) \cup \{s'\}) \\
&= word^l_{M'}((\sigma(s_1, e) \cup \sigma(s_2, e)) \setminus \{s_1, s_2\}) \cup word^l_{M'}(s') \\
&= word^l_M((\sigma(s_1, e) \cup \sigma(s_2, e)) \setminus \{s_1, s_2\}) \cup word^l_{M'}(s') \\
&= (word^l_M(\sigma(s_1, e)) \setminus word^l_M(s_1)) \cup word^l_M(s_1) \\
&= word^l_M(\sigma(s_1, e))
\end{aligned}
$$

Thus, the first statement also holds.

For the second statement, from Lemma 8, it also suffices to show $word^l_M(\sigma(s, e)) = word^l_{M'}(\sigma(s, e))$. If $s_1, s_2 \notin \sigma(s, e)$, then $\sigma'(s, e) = \sigma(s, e)$. From the inductive assumption (the second statement), we have $word^l_M(\sigma(s, e)) = word^l_{M'}(\sigma(s, e))$. Thus, the second statement holds for this case.

Otherwise, if s_1 or s_2 in $\sigma(s, e)$, then $\sigma'(s, e) = (\sigma(s, e) \setminus \{s_1, s_2\}) \cup \{s'\}$. Similar to the proof of the first statement, we have the following equation:

$$
\begin{aligned}
&word^l_{M'}(\sigma(s, e)) \\
&= word^l_{M'}((\sigma(s, e) \setminus \{s_1, s_2\}) \cup \{s'\}) \\
&= (word^l_M(\sigma(s, e)) \setminus word^l_M(s_1)) \cup word^l_{M'}(s')
\end{aligned}
$$

$$= (word^l_M(\sigma(s,e)) \setminus word^l_M(s_1)) \cup word^l_M(s_1)$$
$$= word^l_M(\sigma(s,e))$$

Thus, the second statement also holds. □

References

[1] A. Biermann, J. Feldman, On the synthesis of finite-state machines from samples of their behavior, IEEE Trans. Comput. C-21 (6) (1972) 592–597.
[2] A.V. Raman, J.D. Patrick, P. North, The sk-strings method for inferring PFSA, in: Proceedings of the Workshop on Automata Induction, Grammatical Inference and Language Acquisition at the 14th International Conference on Machine Learning, 1997.
[3] G. Ammons, R. Bodík, J.R. Larus, Mining specifications, in: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '02, ACM, New York, NY, USA, 2002, pp. 4–16.
[4] D. Lo, L. Mariani, M. Pezzè, Automatic steering of behavioral model inference, in: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM, 2009, pp. 345–354.
[5] C. Ghezzi, M. Pezzè, M. Sama, G. Tamburrelli, Mining behavior models from user-intensive web applications, in: Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 277–287.
[6] I. Beschastnikh, Y. Brun, M.D. Ernst, A. Krishnamurthy, Inferring models of concurrent systems from logs of their behavior with CSight, in: Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 468–479.
[7] P.M. Comparetti, G. Wondracek, C. Kruegel, E. Kirda, Prospex: protocol specification extraction, in: 2009 30th IEEE Symposium on Security and Privacy, IEEE, 2009, pp. 110–125.
[8] C. Lee, F. Chen, G. Roşu, Mining parametric specifications, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, New York, NY, USA, 2011, pp. 591–600.
[9] S. Wang, D. Lo, L. Jiang, S. Maoz, A. Budi, Scalable parallelization of specification mining using distributed computing, in: C. Bird, T. Menzies, T. Zimmermann (Eds.), The Art and Science of Analyzing Software Data, Morgan Kaufmann, Boston, 2015, pp. 623–648.
[10] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[11] F. Thollard, P. Dupont, C.D.L. Higuera, Probabilistic DFA inference using Kullback–Leibler divergence and minimality, in: Proceedings of the 17th International Conference on Machine Learning, ICML '00, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000, pp. 975–982.
[12] C. Luo, F. He, C. Ghezzi, Inferring software behavioral models with MapReduce, in: Dependable Software Engineering: Theories, Tools, and Applications, in: Lect. Notes Comput. Sci., vol. 9409, Springer International Publishing, 2015, pp. 135–149.
[13] K.-H. Lee, Y.-J. Lee, H. Choi, Y.D. Chung, B. Moon, Parallel data processing with MapReduce: a survey, SIGMOD Rec. 40 (4) (2012) 11–20.
[14] S. Lamprier, T. Ziadi, N. Baskiotis, L.M. Hillah, Exact and efficient temporal steering of software behavioral model inference, in: 2014 19th International Conference on Engineering of Complex Computer Systems (ICECCS), 2014, pp. 166–175.
[15] J.E. Cook, A.L. Wolf, Discovering models of software processes from event-based data, ACM Trans. Softw. Eng. Methodol. 7 (3) (1998) 215–249.
[16] S.M. Blackburn, R. Garner, C. Hoffman, A.M. Khan, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The DaCapo benchmarks: Java benchmarking development and analysis, in: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2006, pp. 169–190.
[17] D. Lo, S.-C. Khoo, QUARK: empirical assessment of automaton-based specification miners, in: Proceedings of the 13th Working Conference on Reverse Engineering, IEEE, 2006, pp. 51–60.
[18] S.M. Blackburn, R. Garner, C. Hoffman, A.M. Khan, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The DaCapo benchmarks: Java benchmarking development and analysis, in: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2006, pp. 169–190.
[19] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, M.D. Ernst, 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, ACM, 2011, pp. 267–277.
[20] J. Oncina, P. Garcia, Identifying regular languages in polynomial time, in: Advances in Structural and Syntactic Pattern Recognition, in: Series in Machine Perception and Artificial Intelligence, vol. 5, World Scientific, 1992, pp. 99–108.
[21] K.J. Lang, B.A. Pearlmutter, R.A. Price, 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 '98, Springer-Verlag, London, UK, 1998, pp. 1–12.
[22] D. Lo, S.-C. Khoo, SMArTIC: towards building an accurate, robust and scalable specification miner, in: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2006, pp. 265–275.
[23] L. Mariani, M. Pezzè, Dynamic detection of cots component incompatibility, IEEE Softw. 24 (5) (2007) 76–85.
[24] W. Li, A. Forin, S.A. Seshia, Scalable specification mining for verification and diagnosis, in: Proceedings of the 47th Design Automation Conference, DAC '10, ACM, New York, NY, USA, 2010, pp. 755–760.
[25] M.J.H. Heule, S. Verwer, Exact DFA identification using SAT solvers, in: Proceedings of the 10th International Conference on Grammatical Inference: Theoretical Results and Applications, Springer, Berlin, Heidelberg, 2010, pp. 66–79.
[26] M.J.H. Heule, S. Verwer, Software model synthesis using satisfiability solvers, Empir. Softw. Eng. 18 (4) (2012) 825–856.
[27] I. Beschastnikh, Y. Brun, J. Abrahamson, M.D. Ernst, A. Krishnamurthy, Unifying FSM-inference algorithms through declarative specification, in: Proceedings of the 35th International Conference on Software Engineering, 2013, pp. 252–261.
[28] N. Busany, S. Maoz, Behavioral log analysis with statistical guarantees, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 898–901.
[29] D. Angluin, Learning regular sets from queries and counterexamples, Inf. Comput. 75 (2) (1987) 87–106.
[30] P. Dupont, B. Lambeau, C. Damas, A.v. Lamsweerde, The QSM algorithm and its application to software behavior model induction, Appl. Artif. Intell. 22 (1–2) (2008) 77–115.
[31] N. Walkinshaw, K. Bogdanov, M. Holcombe, S. Salahuddin, Reverse engineering state machines by interactive grammar inference, in: 14th Working Conference on Reverse Engineering, 2007, WCRE 2007, 2007, pp. 209–218.
[32] F. Howar, B. Steffen, M. Merten, Automata learning with automated alphabet abstraction refinement, in: Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation, Springer, Berlin, Heidelberg, 2011, pp. 263–277.
[33] N. Walkinshaw, K. Bogdanov, Inferring finite-state models with temporal constraints, in: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, 2008, pp. 248–257.
[34] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in: Computer Aided Verification, Springer, 2000, pp. 154–169.

[35] T.D.B. Le, X.B.D. Le, D. Lo, I. Beschastnikh, Synergizing specification miners through model fissions and fusions (t), in: Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering, 2015, pp. 115–125.
[36] L. Mariani, F. Pastore, Automated identification of failure causes in system logs, in: Proceedings of the 19th International Symposium on Software Reliability Engineering, 2008, pp. 117–126.
[37] J. Whaley, M.C. Martin, M.S. Lam, Automatic extraction of object-oriented component interfaces, Softw. Eng. Notes 27 (4) (2002) 218–228.
[38] R. Alur, P. Černý, P. Madhusudan, W. Nam, Synthesis of interface specifications for Java classes, SIGPLAN Not. 40 (1) (2005) 98–109.
[39] A. Wasylkowski, A. Zeller, C. Lindig, Detecting object usage anomalies, in: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering, ESEC-FSE '07, ACM, New York, NY, USA, 2007, pp. 35–44.
[40] S. Shoham, E. Yahav, S.J. Fink, M. Pistoia, Static specification mining using automata-based abstractions, IEEE Trans. Softw. Eng. 34 (5) (2008) 651–666.
[41] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the 4th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL '77, ACM, New York, NY, USA, 1977, pp. 238–252.
[42] L. Mariani, M. Pezzè, O. Riganelli, M. Santoro, SEIM: static extraction of interaction models, in: Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems, PESOS '10, ACM, New York, NY, USA, 2010, pp. 22–28.
[43] G. de Caso, V. Braberman, D. Garbervetsky, S. Uchitel, Automated abstractions for contract validation, IEEE Trans. Softw. Eng. 38 (1) (2012) 141–162.
[44] I. Krka, Y. Brun, N. Medvidovic, Automatic mining of specifications from invocation traces and method invariants, in: Proceedings of the 22nd ACM International Symposium on Foundations of Software Engineering, FSE 2014, ACM, New York, NY, USA, 2014, pp. 178–189.
[45] V. Dallmeier, C. Lindig, A. Wasylkowski, A. Zeller, Mining object behavior with ADABU, in: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis, WODA '06, ACM, New York, NY, USA, 2006, pp. 17–24.
[46] D. Lorenzoli, L. Mariani, M. Pezzè, Automatic generation of software behavioral models, in: Proceedings of the 30th International Conference on Software Engineering, ICSE '08, ACM, New York, NY, USA, 2008, pp. 501–510.
[47] H. Xiao, J. Sun, Y. Liu, S.W. Lin, C. Sun, TzuYu: learning stateful typestates, in: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering, 2013, pp. 432–442.
[48] T. Xie, D. Notkin, Automatic extraction of object-oriented observer abstractions from unit-test executions, in: Proceedings of the 6th International Conference on Formal Engineering Methods, Springer, Berlin, Heidelberg, 2004, pp. 290–305.
[49] N. Walkinshaw, K. Bogdanov, S. Ali, M. Holcombe, Automated discovery of state transitions and their functions in source code, Softw. Test. Verif. Reliab. 18 (2) (2008) 99–121.
[50] C. Ghezzi, A. Mocci, M. Monga, Synthesizing intensional behavior models by graph transformation, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 430–440.
[51] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, A. Zeller, Generating test cases for specification mining, in: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10, ACM, New York, NY, USA, 2010, pp. 85–96.
[52] T. Berg, B. Jonsson, H. Raffelt, Regular inference for state machines with parameters, in: Proceedings of the 9th Fundamental Approaches to Software Engineering, Springer, Berlin, Heidelberg, 2006, pp. 107–121.
[53] M.D. Ernst, J. Cockrell, W.G. Griswold, D. Notkin, Dynamically discovering likely program invariants to support program evolution, in: Proceedings of the 21st International Conference on Software Engineering, ICSE '99, ACM, New York, NY, USA, 1999, pp. 213–224.
[54] D. Giannakopoulou, Z. Rakamarić, V. Raman, Symbolic learning of component interfaces, in: Proceedings of the 19th International Symposium on Static Analysis, Springer, Berlin, Heidelberg, 2012, pp. 248–264.
[55] F. Howar, D. Giannakopoulou, Z. Rakamarić, Hybrid learning: interface generation through static, dynamic, and symbolic analysis, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, ACM, New York, NY, USA, 2013, pp. 268–279.
[56] B. Schölkopf, C.J. Burges, Advances in Kernel Methods: Support Vector Learning, MIT Press, 1999.
[57] N. Walkinshaw, R. Taylor, J. Derrick, Inferring extended finite state machine models from software executions, Empir. Softw. Eng. (2015) 1–43.
[58] K. Li, R. Groz, M. Shahbaz, Integration testing of distributed components based on learning parameterized i/o models, in: Proceedings of the 26th International Conference on Formal Techniques for Networked and Distributed Systems, Springer, Berlin, Heidelberg, 2006, pp. 436–450.
[59] F. Aarts, B. Jonsson, J. Uijen, Generating models of infinite-state communication protocols using regular inference with abstraction, in: Proceedings of the 22nd International Conference on Testing Software and Systems, Springer, Berlin, Heidelberg, 2010, pp. 188–204.
[60] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, F. Vaandrager, Automata learning through counterexample guided abstraction refinement, in: FM 2012: Formal Methods, Springer, 2012, pp. 10–27.
[61] F. Howar, M. Isberner, B. Steffen, O. Bauer, B. Jonsson, Inferring semantic interfaces of data structures, in: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, Springer, 2012, pp. 554–571.
[62] F. Howar, B. Steffen, B. Jonsson, S. Cassel, Inferring canonical register automata, in: Verification, Model Checking, and Abstract Interpretation, Springer, 2012, pp. 251–266.
[63] S. Cassel, F. Howar, B. Jonsson, B. Steffen, Learning extended finite state machines, in: Software Engineering and Formal Methods, Springer, 2014, pp. 250–264.
[64] R. Medhat, S. Ramesh, B. Bonakdarpour, S. Fischmeister, A framework for mining hybrid automata from input/output traces, in: Proceedings of the 12th International Conference on Embedded Software, IEEE Press, 2015, pp. 177–186.
[65] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, Y. Brun, Behavioral resource-aware model inference, in: Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2014, pp. 19–30.
[66] B. Barre, M. Klein, M. Soucy-Boivin, P.-A. Ollivier, S. Hallé, MapReduce for parallel trace validation of LTL properties, in: Runtime Verification, in: Lect. Notes Comput. Sci., vol. 7687, Springer, Berlin, Heidelberg, 2013, pp. 184–198.
[67] D. Bianculli, C. Ghezzi, S. Krstić, Trace checking of metric temporal logic with aggregating modalities using MapReduce, in: Software Engineering and Formal Methods, in: Lect. Notes Comput. Sci., vol. 8702, Springer International Publishing, 2014, pp. 144–158.
[68] R. Koymans, Specifying real-time properties with metric temporal logic, Real-Time Syst. 2 (4) (1990) 255–299.
[69] M.M. Bersani, D. Bianculli, C. Ghezzi, S. Krstić, P. San Pietro, Efficient large-scale trace checking using MapReduce, in: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, ACM, New York, NY, USA, 2016, pp. 888–898.
[70] D. Basin, G. Caronni, S. Ereth, M. Harvan, F. Klaedtke, H. Mantel, Scalable offline monitoring of temporal specifications, Form. Methods Syst. Des. (2016) 1–34.
[71] D. Basin, F. Klaedtke, S. Müller, Policy monitoring in first-order temporal logic, in: Computer Aided Verification, in: Lect. Notes Comput. Sci., vol. 6174, Springer, Berlin, Heidelberg, 2010, pp. 1–18.