Inferring Software Behavioral Models with MapReduce

Chen Luo[1,2,3], Fei He[1,2,3] and Carlo Ghezzi[4]
[1] Tsinghua National Laboratory for Information Science and Technology (TNList)
[2] Key Laboratory for Information System Security, Ministry of Education
[3] School of Software, Tsinghua University, Beijing 100084, China
[4] Politecnico di Milano, Italy
luoc13@mails.tsinghua.edu.cn, hefei@tsinghua.edu.cn, carlo.ghezzi@polimi.it

ABSTRACT

Software systems are often built without developing any explicit model and therefore research has been focusing on automatic inference of models by applying machine learning 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 focuses on inference of *behavioral models* and explore to use of MapReduce to deal with large logs. The approach consists of two distributed algorithms that perform *trace slicing* and *model synthesis*. For each job, a distributed algorithm using MapReduce is developed. With the parallel data processing capacity of MapReduce, the problem of inferring behavioral 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.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications

General Terms

Algorithms, Verification, Reliability

Keywords

Model Inference, Parametric Trace, Log Analysis, MapReduce

1. INTRODUCTION

Software behavioral 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 first and then used to derive an implementation, reality shows that often models do not exist, or they are inconsistent with the implementation. In fact, building a proper model is hard and requires both mathematical skills and ingenuity. Moreover, even if they are developed, they are often not kept in sync with changes to the implementation.

One promising approach to tackle this problem is to use machine learning to infer software behavioral models automatically from execution logs [8,17]. Many model inference algorithms [5,11,16] 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 large. For example, the production systems in Google generate billions of log events each day [22], which far exceeds the capacity of a single computer.

It is thus desirable to parallelize the processing of massive logs. In prior work [12], Lee et al. proposed an algorithm for slicing traces by parametric events. This algorithm is useful for log processing and model inference. A natural idea to parallelize this algorithm is to divide the trace into multiple segments, process each segment in one node, and then merge all results together. However, this naive solution could lead to incorrect results since events in different segments may be correlated and should be processed together (Section 4).

To handle this, we propose to use MapReduce [10] to deal with large logs in model inference tasks. Using the MapReduce model, we can effectively distribute the processing of massive logs to numerous computing nodes, meanwhile ensuring the related events are always processed together. With the powerful data processing capacity of MapReduce, the problem of inferring behavioral models from large logs can be efficiently solved.

In a nutshell, our approach consists of two stages: *trace slicing* and *model synthesis*. The first stage parses and slices the log into different trace slices, and constructs a prefix tree acceptor as the intermediate result. The second stage reads the prefix tree acceptor, and synthesizes the behavioral model. Both stages are realized under the MapReduce framework. We develop a distributed algorithm for the trace slicing and model synthesis, respectively. With these two algorithms, we propose a novel MapReduce framework for inferring software behavioral models.

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 method that, to the best of our knowledge, represents a novel attempt to use the MapReduce framework for inferring software behavioral models;
- We implemented a prototype of our technique. The experimental results show the promising performance of our approach.

This paper extends our previous work [18] that appeared in SETTA with several extensions. First, we developed several practical optimizations to further improve our approach. Moreover, we formally proved the correctness of our distributed trace slicing and distributed model synthesis algorithms respectively. We also conducted more thorough experiments to evaluate 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 some formal definitions of this work. Section 4 and Section 5 introduce our distributed algorithms for trace slicing and model synthesis, respectively. Section 6 reports the experimental results. Section 7 describes some possible extensions. Section 8 discusses the related work and Section 9 concludes this paper.

2. APPROACH OVERVIEW

2.1 MapReduce

MapReduce [10] is a large-scale parallel data processing framework based on distributed architectures. It hides the details of data distribution, load balancing, and failure recovery while provides simple yet powerful interfaces to users. Due to its simplicity, MapReduce has become one of the most popular distributed computing frameworks. Hadoop [1] is an 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:

$$\text{MAP}: \quad (k1, v1) \rightarrow \ list(k2, v2)$$
$$\text{REDUCE}: \quad (k2, list(v2)) \rightarrow \ list(k3, v3)$$

The main flow of a MapReduce job is illustrated in Figure 1. The job consists of three phases, i.e., *map*, *shuffle*, and *reduce*. In the *map* phase, the input data are 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 corresponds to 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.

[1] `http://hadoop.apache.org/`

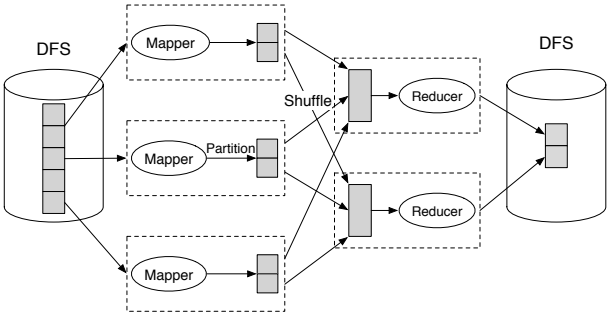


Figure 1: MapReduce overview

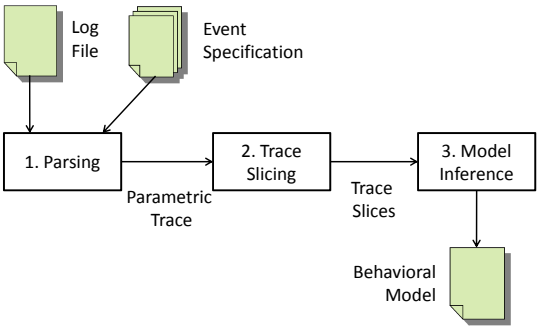


Figure 2: Behavioral model inference overview

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 behavioral model inference algorithm is shown in Figure 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*. A parameter corresponds to an entity in the system. We say an *interaction* happens when two or more events with the same parameter are detected in the log file. After parsing, we get a sequence of parametric events, called a *parametric trace*.

The parametric trace may contain many independent interactions, and thus cannot be directly used for model synthesis. A trace slicer is called to slice the parametric trace into many slices, each of which corresponds to an interaction scenario. Finally, a synthesis algorithm is called to infer the behavioral model from the set of trace slices.

2.3 Running Example

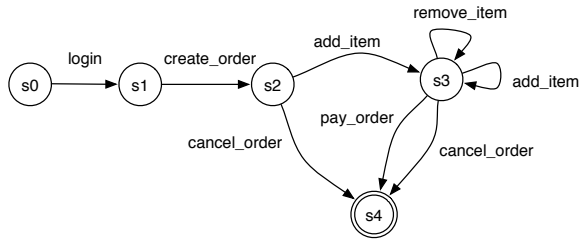
As a running example, consider the on-line shopping system shown in Figure 3. The relevant events and their cor-

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) Relevant events and parameters

```
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 paramtric trace



(c) The behavioral model

Figure 3: An online shopping system example

responding parameters are shown in Figure 3a. Briefly, we are interested in the following events:

- the user *userid logins* 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 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 online shopping system may generate a huge log file. Most information in the log is irrelevant to our concerns. Thus we need a parser to extract relevant events and parameters from the log file. A parametric trace excerpt is shown in Figure 3b. Note that multiple users can operate the shopping system at the same time, and the events about their operations are interleaved in the log file. The behavioral model of the running example is depicted in Figure 3c. However, this model is *unknown* to us.

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 Figure 4, our approach consists of two stages, i.e., the distributed trace slicing stage and the distributed model synthesis stage, both of which are realized using MapReduce. The first stage takes as input a log file, performs the log parsing and trace slicing, and outputs a prefix tree acceptor (PTA) [16]. The log parsing task is performed by

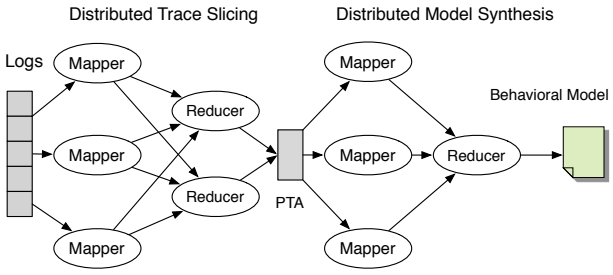


Figure 4: Model inference with MapReduce

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 stage takes as input the PTA generated in the former stage, and infers the behavioral model by a distributed model synthesis algorithm. With the large-scale data processing capacity of MapReduce framework, the problem of inferring behavioral models from large log files can be efficiently solved.

Although the basic algorithms for trace slicing [12] and model synthesis [8] exist, our contribution is to realize a novel MapReduce version of both algorithms and integrate them seamlessly.

3. FORMAL DEFINITIONS

This section introduces the formal definitions needed in our framework. Some of these definitions originate from [12].

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 Figure 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 called* complete *if* $Dom(\theta) = X$*. Let* $Y \subseteq Dom(\theta)$*, a restriction* $\theta \restriction_Y$ *of* θ *to* Y *is a parameter instance such that* $Dom(\theta \restriction_Y) = Y$ *and for any* $y \in Y$*,* $\theta \restriction_Y (y) = \theta(y)$*.*

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 parameters for each base event to be fixed as in [12].

DEFINITION 4. *A* trace *is a finite sequence of base events. A* parametric trace *is a finite sequence of parametric events. Denote $e \in \tau$ (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 Figure 3b gives a parametric event, and the sequence of all parametric events in Figure 3b gives a parametric trace.

DEFINITION 5. *A parameter instance θ' is called* less 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. The parameter instance $\langle user1\rangle$ is less informative than $\langle user1, order1\rangle$, and $\langle user1, order1\rangle$ is less informative than itself.

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

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

Intuitively, the θ-trace slice $\tau \restriction_\theta$ first filters out the irrelevant parametric events to θ, then leaves out the parameter instances and only keeps the base events. For example, let τ_1 be the parametric trace in Figure 3b. For $\theta_1 = \langle user1, order1\rangle$, $\tau_1 \restriction_{\theta_1}$ is the sequence of: *login, create_order, pay_order*. Let $\theta_2 = \langle user1, order1, item1\rangle$, then $\tau_1 \restriction_{\theta_2}$ 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. 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 $user1$ and $order2$ 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* strong 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*

- *$e\langle\theta\rangle \in \tau$, 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. Since the parameter instances $\langle user1, order1\rangle$ and $\langle order1, item1\rangle$ satisfy the first condition in the above definition, and $\langle user1, order1\rangle \sqcup \langle order1, item1\rangle = \langle user1, order1, item1\rangle$, thus the parameter instance $\langle user1, order1, item1\rangle$ is connected. In the remainder of this paper, we consider only trace slices for complete and connected parameter instances to avoid meaningless results as in [12].

4. DISTRIBUTED TRACE SLICING WITH MAPREDUCE

This section presents our distributed trace slicing algorithm with MapReduce. As mentioned before, a naive solution to parallelize the trace slicing task is to divide the trace into multiple segments, process each segment with a standalone slicing algorithm [12], and then merge the sliced results together. However, this solution could lead to incorrect results. For example, suppose the trace τ_1 in Figure 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. According to Definition 9, there is no complete and connected parameter instance with respect to τ_1^1, which means no slice will be generated for τ_1^1. Thus, simply merging trace slices from τ_1^1 and τ_1^2 leads to incorrect results.

To handle this, instead of simply dividing the original trace into multiple segments, we group all correlated parametric events and send them to the same reducer to generate trace slices. In the following, we first propose a data encoding mechanism, and then introduce the MAP and REDUCE functions. We also discuss the correctness of our distributed slicing algorithm and several practical optimizations.

4.1 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 watch a subset of X, and for each parametric event $e\langle\theta\rangle$, we report the watched value on θ as its key, which is used by MapReduce to choose 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 Figure 3b is $\perp$. And the keys of the remaining parametric events in Figure 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 $T_2 = \{e\langle\theta\rangle | \mathcal{X} \cap \mathcal{D}_e(e) = \emptyset\}$. Continue 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 with the ID of $hash(k_1)$. 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 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.

We now discuss how to choose $\mathcal{X}$ 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 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:

HEURISTIC 1. *The set $\mathcal{X}$ should be chosen such that $\hat{T}_2$ is as small as possible.*

This heuristic 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 definition, which is known a priori. Thus, the parameter window $\mathcal{X}$ can be decided before MapReduce computations.

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 heuristic.

HEURISTIC 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 heuristic to maximize $\hat{T}_2$. If there is more than one candidate $\mathcal{X}$, we then apply the second heuristic to select the one with the largest size. For example, the parameter window for the running example is chosen as $\mathcal{X} = \{orderid\}$ according to the event definition in Figure 3a.

```
1: function MAP(line) ▷ /* X: the parameter window */
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⟩);
```

Figure 5: Trace slicing: Map function

4.2 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.

Figure 5 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.

Consider the running example with $\mathcal{X} = \{orderid\}$, and assume there are two mappers. Then the parametric events labeled from 1 to 5 in Figure 3b are handled by $Mapper_1$, and the parametric events labeled from 6 to 9 in Figure 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 the convenience of 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 Figure 6. 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.

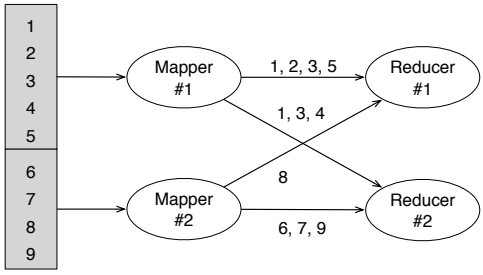


Figure 6: Trace slicing: the running example

```
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_tmp ← ∅;
12:     for e⟨θ⟩ ∈ values[] do
13:         if θ ∉ Dom(Δ_tmp) then
14:             Initialize the list Δ_tmp(θ);
15:         Insert e into Δ_tmp(θ);
16:     return Δ_tmp;

17: function CONSTRUCT(Δ)
18:     Ω ← Dom(Δ);
19:     while ∃θ_1, θ_2 ∈ Ω s.t. θ_1 ⋈ θ_2 ∧ (θ_1 ⊔ θ_2 ∉ Ω) do
20:         Ω ← Ω ∪ {θ_1 ⊔ θ_2};
21:     for complete parameter instance θ ∈ Ω do
22:         Γ ← {Δ(θ')|θ' ⊑ θ and θ' ∈ Dom(Δ)};
23:         Compute τ ↾_θ by merging event lists in Γ;
24:         Update PTA using τ ↾_θ;
```

Figure 7: Trace slicing: Reduce function

4.3 Reducer

Recall that the shuffle phase, MapReduce partitions and sorts key-value pairs to ensure that values corresponding to the same key are passed to a single reduce call. Denote *values*[] the list of parametric events with the key of *key*. The REDUCE function is called for each pair of *key* and *values*[]. Consider $Reducer_1$ in Figure 6, 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 Figure 7. Note that the MapReduce framework is configured so that the events in T_2, i.e., $key = \perp$, would always come first. For the key $\perp$, we call the RESTORE function (lines 2-4) to reorganize *values*[] into several lists (lines 11-16). Each list $\Delta_{tmp}(\theta)$ corresponds to a parameter instance θ, and consists of base events only. Here we abuse the notion of $Dom(\Delta_{tmp})$ (line 13), which denotes the set of parameter instances θ where the list $\Delta_{tmp}(\theta)$ is defined, i.e., $Dom(\Delta_{tmp}) = \{\theta|\Delta_{tmp}(\theta) \text{ is defined}\}$. Recall that each event is associated with a *timestamp*. At line 15, the base event e is inserted to a proper position in $\Delta_{tmp}(\theta)$ such that $\Delta_{tmp}(\theta)$ is in ascending order of *timestamp*.

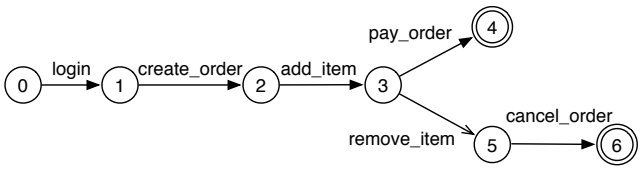


Figure 8: PTA for the running example

Then for keys other than $\perp$, we also call the RESTORE function to reorganize *values*[] into several lists Δ (line 5). Note that $\Delta_\perp$ is global and shared by multiple calls of the REDUCE function. As a result, when the REDUCE function proceeds to line 5, $\Delta_\perp$ must have already been initialized. The while loop at line 6 tries to retrieve some lists $\Delta_\perp(\theta_\perp)$ into Δ such that $\theta_\perp$ can be combined with some $\theta \in Dom(\Delta)$. According to Definition 9, if $\theta_\perp$ and θ are connected, and $\theta_\perp \bowtie \theta$, then $\theta_\perp \sqcup \theta$ is also *connected*. Thus, the list $\Delta_\perp(\theta_\perp)$ is added into Δ (line 8). Note that $\theta_\perp$ may again be strong compatible with other parameter instances in T_2; this process is thus iterative.

Finally, the CONSTRUCT function is called at line 9 to compute trace slices and then update the intermediate structure *PTA*. Ω is the set of parameter instances in Δ. The function first tries to combine (lines 19-20) all strong compatible parameter instances in Ω. This process is iterative, since the newly generated parameter instance may be combined with the existing ones. Then the trace slice for each complete and connected parameter instance θ is constructed by merging the event lists of θ's less informative parameter instances in ascending order of *timestamp*(lines 22-24).

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 $\langle user1\rangle$ are strong 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 $\tau \upharpoonright_\theta = login, create\_order, add\_item, pay\_order$.

We take the prefix tree acceptor (PTA) as the intermediate structure. Each reducer keeps a PTA. The generated trace slices (at line 24) 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 Section 5) 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, and an example is shown in Figure 8.

4.4 Correctness

In the following, we show the correctness of our distributed trace slicing algorithm. For the ease of discussion, given a

parameter instance θ and a parametric trace τ, we denote $\theta \in \tau$ if there exists a parametric event $e\langle\theta\rangle \in \tau$.

First, we show some properties of the connected parameter instance (Definition 9).

LEMMA 2. *Given a parametric trace τ and a connected parameter instance θ, there exists a sequence of parameter instances $\theta_1, \ldots, \theta_n \in \tau$ such that:*

- *$\theta = \theta_1 \sqcup \ldots \sqcup \theta_n$, and*
- *for any θ_i and θ_j in the sequence, θ_j is* reachable *from θ_i, i.e., there exist $\theta'_1, \ldots, \theta'_k$ 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 $\theta \in \tau$, then the sequence only consists of θ, since $\theta = \theta$ and $\theta \bowtie \theta$. Thus, the lemma holds.

Inductive step: suppose there exist θ_1 and θ_2 such that both θ_1 and θ_2 are 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 $\theta_2 = \theta_2^1 \sqcup \ldots \sqcup \theta_2^v$. 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 θ_i and θ_j be two parameter instances in $\theta_1^1, \ldots \theta_1^u, \theta_2^1, \ldots \theta_2^v$. From the inductive assumption, we only need to consider the case where θ_i in $\theta_1^1, \ldots, \theta_1^u$ and θ_j in $\theta_2^1, \ldots, \theta_2^v$, since otherwise, the statement trivially holds. 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 parameter 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 3. *Given a parametric trace τ and a parameter window $\mathcal{X}$, for any complete and connected parameter instance θ, there exists a parameter instance $\theta' \in \tau$ such that $\theta' \restriction_{\mathcal{X}} = \theta \restriction_{\mathcal{X}}$.*

PROOF. From Lemma 2, since θ is connected, there exists a sequence of parameter instances $\theta'_1, \ldots \theta'_n \in \tau$ such that $\theta = \theta'_1 \sqcup \ldots \sqcup \theta'_n$. Moreover, from the definition of $\mathcal{X}$ (Definition 10), for any θ'_i 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 θ' in the sequence such that $\mathcal{X} \subseteq Dom(\theta')$. Then from the definition of the combination operator $\sqcup$ (Definition 7), we have $\theta' \restriction_{\mathcal{X}} = \theta \restriction_{\mathcal{X}}$. □

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

LEMMA 4. *Given a parametric trace τ and a complete and connected parameter instance θ, for any parameter instance $\theta' \in \tau$ such that $\theta' \sqsubseteq \theta$, $\Delta(\theta')$ is defined after the* while *loop at line 6 of the* REDUCE *function for key $= \theta \restriction_{\mathcal{X}}$.*

PROOF. Let *value*[] be the list of parametric events for $key = \theta \restriction_{\mathcal{X}}$, and $e'\langle\theta'\rangle$ be a parametric event in τ such that $\theta' \sqsubseteq \theta$. If $\mathcal{X} \subseteq Dom(\theta')$, which means $key(e'\langle\theta'\rangle) = \theta' \restriction_{\mathcal{X}} = \theta \restriction_{\mathcal{X}}$, then $e'\langle\theta'\rangle \in values[]$. Thus, $\Delta(\theta')$ is defined after line 5 of the REDUCE function for $key = \theta \restriction_{\mathcal{X}}$, 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 2, since θ is connected, there exists a sequence of parameter instances $\theta_1 \ldots \theta_n \in \tau$ such that $\theta = \theta_1 \sqcup \ldots \sqcup \theta_n$. Then from the proof of Lemma 3 and without loss of generality, let θ_1 in the sequence be the parameter instance such that $\theta \restriction_{\mathcal{X}} = \theta_1 \restriction_{\mathcal{X}}$. From the previous discussion, $\Delta(\theta_1)$ is defined after line 5 of the REDUCE function for $key = \theta \restriction_{\mathcal{X}}$.

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 parameter instance θ_i in the sequence 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 2, there exist $\theta'_1, \ldots, \theta'_k$ in the sequence of $\theta_1, \ldots, \theta_n$ such that $\theta_1 \bowtie \theta'_1, \ldots, \theta'_{k-1} \bowtie \theta'_k$ and $\theta'_k \bowtie \theta_i$. By concatenating the above sequence 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 $key = \theta \restriction_{\mathcal{X}}$ ($\mathcal{X} \subseteq Dom(\bar{\theta})$), 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 $key = \theta \restriction_{\mathcal{X}}$, $\Delta_\perp(\theta')$ 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 $\tau \restriction_\theta$ is correctly constructed at line 23, which is achieved by the following lemma.

LEMMA 5. *$\tau \restriction_\theta$ computed at line 22 satisfies the definition of θ-trace slice of τ (Definition 6).*

PROOF. From the definition of trace slice (Definition 6), given a parameter 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 4, for any $e'\langle\theta'\rangle \in \tau$ such that $\theta' \sqsubseteq \theta$, $\Delta(\theta')$ is defined at line 22 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 23. □

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 23, the parameter instance θ is complete and connected, and*
2. *for any complete and connected parameter instance θ, θ-trace slice of τ is constructed at line 23.*

PROOF. The first statement trivially holds since the guards at line 19 and 21 ensure only $\tau \restriction_\theta$ for complete and connected parameter instances are constructed.

We then mainly focus on the second statement. From Lemma 3, for any complete and connected parameter instance θ, there exists a parametric event $e'\langle\theta'\rangle \in \tau$ such

that $\theta' \upharpoonright_{\mathcal{X}} = \theta \upharpoonright_{\mathcal{X}}$, which means there exists a key-value pair output by mappers with $key = \theta \upharpoonright_{\mathcal{X}}$. Then we only need to show θ-trace slice of τ can be constructed in the REDUCE function for $key = \theta \upharpoonright_{\mathcal{X}}$, which is equivalent to show $\theta \in \Omega$ at line 21 from Lemma 5. From Lemma 2, since θ is connected, there exists a sequence of parameter instances $\theta_1, \ldots \theta_n \in \tau$ such that $\theta = \theta_1 \sqcup \ldots \sqcup \theta_n$, and for any θ_i and θ_j in the sequence, θ_j is reachable from θ_i. Moreover, from Lemma 4, 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 parameter instances $\theta_1, \ldots \theta_n$, and the second statement holds. □

4.5 Optimizations

We have developed several optimizations to improve our distribute 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 parameter instance into a list.

Dictionary compression. Since the base events are known in advance, we build a dictionary to map each base event to an integer (or a byte if the number of base events is less than 255) to reduce the intermediate data transmission. During the map phase, each mapper simply looks up the dictionary and outputs the corresponding integer instead of the original base event. After the final model is synthesized (Definition 5), the encoded base events are decoded with the dictionary for representation.

Parameter instance partitions. In the reducer, we partition all parameter instances into groups based on their domains, i.e., two parameter instances θ_1 and θ_2 are in the same group if and only if $Dom(\theta_1) = Dom(\theta_2)$. Thus, combining parameter instances in the same group or two groups where the domain of one group subsumes another will not generate any new parameter instance and can be skipped during the while loop at line 19 of Figure 7.

Inverted index optimization. Since $\Delta_\perp$ is unchanged after being initialized, we build an inverted index for each parameter instance group in $\Delta_\perp$. For each value v_x of a parameter x, we record the set of parameter 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 strong compatible parameter instances in $\Delta_\perp$ for a given parameter instance θ can be greatly reduced using this inverted index. Specially, for each parameter 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 parameter instances, each of which shares the same parameter value with θ on the parameter x. Thus, the set of strong compatible instances for θ in this group is obtained by intersecting these sets of parameter instances for all shared parameters.

5. DISTRIBUTED MODEL SYNTHESIS WITH MAPREDUCE

Once the complete PTA has been generated, as previously shown, many off-the-shelf model synthesis algorithms [8, 19, 20] 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 [8] with MapReduce to improve efficiency.

Most existing model synthesis algorithms can be viewed as variants of k-tail, which take as input a PTA, then repetitively 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 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.

5.1 Data Encoding

To realize the distributed model synthesis algorithm with MapReduce, the intermediate results should be in the form of key-value pairs. The "value" here is a state, we thus 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 state of these states.

Before discussing how to encode states, we first introduce some notations of the behavioral model. A behavioral 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, and is called as a *k-word*. Given an automaton M, let f be a function from $S \times \Sigma^*$ to Boolean such that for any state $s \in S$ and any word $\omega \in \Sigma^*$, $f(s, \omega) = 1$ iff starting from s, the word ω is *accepted* by σ^* [2].

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 $\omega \in \Sigma^{\leq k}$, $f(s_1, \omega) = 1$ iff $f(s_2, \omega) = 1$.*

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 $[s]$.

[2]We do not require that a word ends in a final state, as in [9].

```
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;
```

Figure 9: Distributed model synthesis

LEMMA 6. *For any two 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 $sig[i] = 1$ iff with the i-th k-word ω in $\Sigma^{\leq k}$, $f(s,\omega) = 1$ for $1 \leq i \leq |\Sigma^{\leq k}|$.*

By Lemma 6, 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.

5.2 Mapper and Reducer

The pseudocode of distributed model synthesis is shown in Figure 9. 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} in correspondence to 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.

5.3 Correctness

In the subsection, we discuss the correctness of our model synthesis algorithm, i.e., the model generated by our distributed algorithm is identical to the one output by the original algorithm. Since we merge k-equivalent states in parallel, we only need to show 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 $word_M^k(s) = \{\omega | \omega \in \Sigma^{\leq k} \wedge f(s,\omega) = 1\}$, 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., $word_M^k(S) = \bigcup_{s \in S} word_M^k(s)$. 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 $e \in \Sigma$, $word_M^{k-1}(\sigma(s_1, e)) = word_M^{k-1}(\sigma(s_2, e))$.*

PROOF. First, we show if for any base event $e \in \Sigma$, $word_M^{k-1}(\sigma(s_1, e)) = word_M^{k-1}(\sigma(s_2, e))$, then s_1 and s_2 are k-equivalent. Let $e \in \Sigma$ be a base event. Since $word_M^{k-1}(\sigma(s_1, e)) = word_M^{k-1}(\sigma(s_2, e))$, we have $e \cdot word_M^{k-1}(\sigma(s_1, e)) = e \cdot word_M^{k-1}(\sigma(s_2, e))$. Thus, $\bigcup_{e \in \Sigma} e \cdot word_M^{k-1}(\sigma(s_1, e)) = \bigcup_{e \in \Sigma} e \cdot word_M^{k-1}(\sigma(s_2, e))$, which implies $word_M^k(s_1) = word_M^k(s_2)$, and the statement holds.

Then, we show if s_1 and s_2 are k-equivalent, then for any base event $e \in \Sigma$, $word_M^{k-1}(\sigma(s_1, e)) = word_M^{k-1}(\sigma(s_2, e))$. 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 $word_M^k(s_1) = word_M^k(s_2)$, which implies $e \cdot \omega \in \bigcup_{e' \in \Sigma} e' \cdot word_M^{k-1}(\sigma(s_1, e'))$ iff $e \cdot \omega \in \bigcup_{e' \in \Sigma} e' \cdot$

$word_M^{k-1}(\sigma(s_2, e'))$ Moreover, since $e \cdot \omega \notin \bigcup_{e' \in (\Sigma \setminus \{e\})} e' \cdot word_M^{k-1}(\sigma(s_1, e'))$ and $e \cdot \omega \notin \bigcup_{e \in (\Sigma \setminus \{e\})} e' \cdot word_M^{k-1}(\sigma(s_2, e'))$, we have $e \cdot \omega \in e \cdot word_M^{k-1}(\sigma(s_1, e))$ iff $e \cdot \omega \in e \cdot word_M^{k-1}(\sigma(s_2, e))$, which means $\omega \in word_M^{k-1}(\sigma(s_1, e))$ iff $\omega \in word_M^{k-1}(\sigma(s_2, e))$. Thus, the statement holds. □

The following theorem ensures 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:*

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

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 $word_M^k(s_1) = word_M^k(s_2)$, and we only need to consider $word_M^k(s_1) = word_{M'}^k(s')$. 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, $e \in word_M^1(s_1)$ iff $e \in word_{M'}^1(s')$, 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, $e \in word_M^1(s)$ iff $e \in word_{M'}^1(s)$, 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 $word_M^l(\sigma(s_1, e)) = word_{M'}^l(\sigma'(s', e))$. 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 $word_M^l(\sigma(s_1, e) \cup \sigma(s_2, e)) = word_{M'}^l(\sigma(s_1, e) \cup \sigma(s_2, e))$. Moreover, from Lemma 7, we have s_1 and s_2 are $(l+1)$-equivalent, which further implies $word_M^l(\sigma(s_1, e)) = word_M^l(\sigma(s_2, e))$ from Lemma 8. Then, we have the following equation:

$$\begin{aligned} &word_{M'}^l(\sigma'(s', e)) \\ &= word_{M'}^l(\sigma(s_1, e) \cup \sigma(s_2, e)) \\ &= word_M^l(\sigma(s_1, e)) \cup word_M^l(\sigma(s_2, e)) \\ &= word_M^l(\sigma(s_1, e)) \end{aligned}$$

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 $word_M^l((\sigma(s_1, e) \cup \sigma(s_2, e)) \setminus \{s_1, s_2\}) = word_{M'}^l((\sigma(s_1, e) \cup \sigma(s_2, e)) \setminus \{s_1, s_2\})$. Also from the inductive assumption (the first statement), we have $word_M^l(s_1) = word_M^l(s_2) = word_{M'}^l(s')$. Moreover, from the previous discussion, we have $word_M^l(\sigma(s_1, e)) = word_M^l(\sigma(s_2, e))$. Then, we have the following equation:

$$\begin{aligned} &word_{M'}^l(\sigma'(s', e)) \\ &= word_{M'}^l(((\sigma(s_1, e) \cup \sigma(s_2, e)) \setminus \{s_1, s_2\}) \cup \{s'\}) \\ &= word_{M'}^l((\sigma(s_1, e) \cup \sigma(s_2, e)) \setminus \{s_1, s_2\}) \cup word_{M'}^l(s') \\ &= word_M^l((\sigma(s_1, e) \cup \sigma(s_2, e)) \setminus \{s_1, s_2\}) \cup word_{M'}^l(s') \\ &= (word_M^l(\sigma(s_1, e)) \setminus word_M^l(s_1)) \cup word_M^l(s_1) \\ &= word_M^l(\sigma(s_1, e)) \end{aligned}$$

Thus, the first statement also holds.

For the second statement, from Lemma 8, it also suffices to show $word_M^l(\sigma(s, e)) = word_{M'}^l(\sigma(s, e))$. 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 $word_M^l(\sigma(s, e)) = word_{M'}^l(\sigma(s, e))$. 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:

$$\begin{aligned} &word_{M'}^l(\sigma(s, e)) \\ &= word_{M'}^l((\sigma(s, e) \setminus \{s_1, s_2\}) \cup \{s'\}) \\ &= (word_M^l(\sigma(s, e)) \setminus word_M^l(s_1)) \cup word_{M'}^l(s') \\ &= (word_M^l(\sigma(s, e)) \setminus word_M^l(s_1)) \cup word_M^l(s_1) \\ &= word_M^l(\sigma(s, e)) \end{aligned}$$

Thus, the second statement also holds. □

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, note that the words with the length greater than k accepted by each state may change during the state merging process.

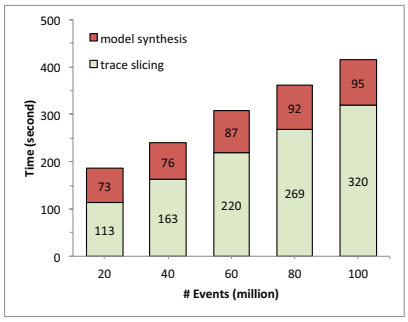
6. EXPERIMENTAL EVALUATION

We implemented our approach on top of Hadoop 1.2.1 [3], and conducted experiments on Amazon Elastic MapReduce clusters [4]. Each computing node in the cluster has a dual-core CPU and 7.5 GB memories. We let each node serve as two mappers and one reducer simultaneously. The running time spent on both MapReduce jobs (trace slicing and model synthesis) is measured separately. Each experiment is performed 3 times, and the average value is reported.

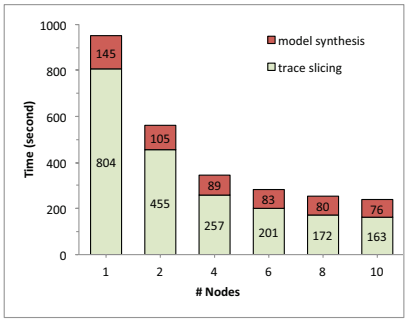
The datasets used in our experiments are randomly generated as follows: (1) An automaton is randomly generated as the target model to be inferred, which contains 50 states and maximally 5 transitions per state. (2) The automaton is randomly simulated to generate parametric traces. Each parametric trace is with 10 to 100 parametric events. The simulation repeats until the total number of events exceeds

[3] `http://hadoop.apache.org/`

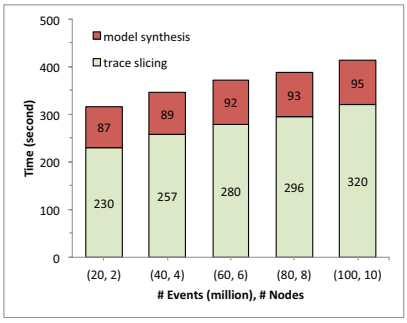
[4] `http://aws.amazon.com/elasticmapreduce/`



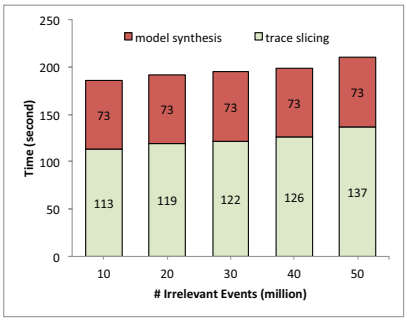
(a) Running time with inreasing log size



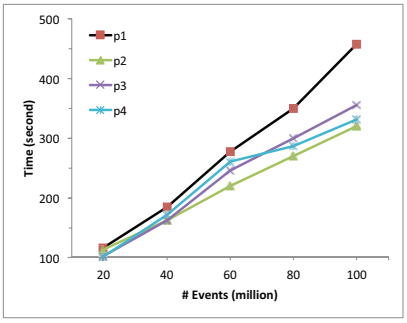
(b) Running time with increasing number of nodes



(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	210/80	217/85	223/94
2	242/82	251/87	254/97
3	265/81	267/84	275/95
4	314/84	317/90	320/98

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

Figure 10: Experimental results

a predefined threshold. (3) All generated parametric traces are randomly mixed up. (4) Irrelevant entries are randomly added to the log as noises. Unless explicitly pointed, other parameters are set as: $|\mathcal{E}| = 15$, $|X| = 4$ and $k = 1$. The event definition $\mathcal{D}_e$ is randomly determined, and the parameter value is randomly chosen from integer domain. By default, the log in each experiment contains the same number of parametric events and irrelevant log entries. The size of the largest log file used in our experiments exceeds 10 GB.

We designed several sets of experiments to evaluate our approach, ranging from basic performance, speed up, scalability to the impact of irrelevant events, chosen of $\mathcal{X}$ and event definitions. The experimental results are reported and discussed below. Note that we do not measure the accuracy of the inferred model, since we mainly focus on the performance and the accuracy of the k-tail algorithm has been well studied in [14]. More accurate models can be inferred by combining our approach with more advanced model inference algorithms as discussed in Section 7.2.

Basic Performance. The first set of experiments tests the running time of our approach for logs with increasing size. Note that the log size is evaluated by the number of events in the log. All logs were produced by our random dataset generator. Sizes of these logs range from 20 to 100 million events. The cluster size is fixed to 10 nodes.

The experimental results are plotted in Figure 10a. Each column in the graph contains two parts, representing the running time of trace slicing and model synthesis, respectively. Most of the running time is spent on trace slicing. The performance of our approach is very promising. The total processing time for the largest log (the file size exceeds 10GB) is less than 7 minutes.

Speed-up. In the second set of experiments, 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 cluster size varies from 1 node to 10 nodes.

The experimental results are plotted in Figure 10b. 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.

Scalability. The third set of experiments 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 Figure 10c. 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.

Impact of Irrelevant Events. The fourth set of experiments 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 Figure 10d.

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 percent of irrelevant events may be very high.

Impact of $\mathcal{X}$. Recall from Section 4.1 that the chosen of $\mathcal{X}$ may affect the performance of the trace slicing job. The fifth set of experiments thus evaluates the impact of $\mathcal{X}$ on the running time of trace slicing and the effectiveness of our heuristics for determining $\mathcal{X}$. In the experiments, X contains four parameters p_1, p_2, p_3 and p_4, and $\mathcal{X}$ is chosen as the singleton set of four parameters, respectively. Note that $\mathcal{X}$ is chosen as $\{p_2\}$ automatically by our heuristics. The cluster size is fixed to 10 nodes, while the log size ranges from 20 million to 100 million events.

The running time is shown in Figure 10e. As we can see, the chosen of $\mathcal{X}$ may have a big impact on the trace slicing job. $\mathcal{X} = \{p_1\}$ corresponds to the most base events in $\hat{T}_2$, and also incurs the longest running time. This experiment also shows our heuristics are effective for determining $\mathcal{X}$, since $\mathcal{X} = \{p_2\}$ performs better compared with other $\mathcal{X}$.

Impact of $|\mathcal{E}|$ and $|X|$. The last set of experiments 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 Figure 10f, and each cell contains 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 $|X|$.

7. DISCUSSION

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

7.1 Invariant Mining

Our approach can be easily 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 [6, 16]. 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 [6]. 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 $a \rightarrow b$, $a \not\rightarrow b$ 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.

7.2 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 inference algorithms [8, 19] share a similar work flow: they first process the given traces and construct a PTA, 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 inference algorithms. The user needs to run the trace slicing job only. Based on the powerful data processing ability of MapReduce, this job can be be performed efficiently. Then the generated PTA and optionally temporal invariants can be fed to the existing algorithms to synthesize the model.

8. RELATED WORKS

The related works fall into two categories: software behavioral model inference and trace checking with MapReduce.

8.1 Software Behavioral Model Inference

A lot of work exists on inferring software behavioral models from execution traces. Ammons et al. [1] first proposed the technique of *specification mining* to mine program specifications from program execution traces. Lo et al. [15] introduced the idea of error-trace filtering and trace clustering to improve the accuracy of the inferred model. GK-Tail [17] extends the k-tail algorithm and infers extended finite state machines. Lee et al. [12] proposed the trace slicing technique to mine parametric specifications, which also inspired our work. Ghezzi et al. [11] inferred users' behavior models from web application logs, and employed probabilistic model checking to analyze users' navigational behaviors.

As mentioned, incorporating temporal invariants can improve the accuracy of inferred models. Walkinshaw and Bogdanov [21] first described this idea by taking LTL constraints as additional input, and used model checking technique to guide the state merging process. Lo et al. [16] improved the work in [21] by inferring statistically significant temporal invariants and used the inferred invariants to guide the state merging step. Moreover, the tool Synoptic [6] automatically mines temporal invariants from execution traces and uses refinement and coarsening to generate accurate but concise models. CSight [5] extends Synoptic to support model inference of concurrent systems.

However, to the best of our knowledge, there is no previously published work on applying MapReduce to model inference.

8.2 Trace Checking with MapReduce

Recently, there have been several works on checking trace compliance against temporal logics using MapReduce. Barre et al. [2] presented an algorithm for checking Linear Temporal Logic (LTL) formula over event traces with MapReduce. The algorithm evaluates a LTL formula iteratively in a bottom-up fashion, and in each iteration, all subformulae with the same depth are evaluated in a MapReduce job. Bianculli et al. [7] further improved the work [2] by supporting specifications expressed in metric temporal logic with

aggregating modalities. Basin et al. [3] presented a formal log slicing framework for checking policies expressed with Metric First-Order Temporal Logic (MFOTL) [4]. In their work, a log is represented as a temporal structure, which can be sliced by data or time.

These works share some similarities with ours, i.e., log processing with MapReduce. But the major difference is that our work focus on behavioral model inference from large logs, rather than checking compliance against temporal logics.

9. CONCLUSION

In this paper, we presented an approach to infer software behavioral model 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.

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 precise and robust model inference algorithms such as sk-strings [19] or incorporating temporal invariants [16] during inference phase.

10. REFERENCES

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 4–16, New York, NY, USA, 2002. ACM.

[2] B. Barre, M. Klein, M. Soucy-Boivin, P.-A. Ollivier, and S. Hallé. Mapreduce for parallel trace validation of ltl properties. In S. Qadeer and S. Tasiran, editors, *Runtime Verification*, volume 7687 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin Heidelberg, 2013.

[3] D. Basin, G. Caronni, S. Ereth, M. Harvan, F. Klaedtke, and H. Mantel. Scalable offline monitoring. In B. Bonakdarpour and S. Smolka, editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 31–47. Springer International Publishing, 2014.

[4] D. Basin, F. Klaedtke, and S. Müller. Policy monitoring in first-order temporal logic. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2010.

[5] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering*, pages 468–479. ACM, 2014.

[6] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and 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*, pages 267–277. ACM, 2011.

[7] D. Bianculli, C. Ghezzi, and S. Krsti. Trace checking of metric temporal logic with aggregating modalities using mapreduce. In D. Giannakopoulou and G. Salaün, editors, *Software Engineering and Formal Methods*, volume 8702 of *Lecture Notes in Computer Science*, pages 144–158. Springer International Publishing, 2014.

[8] A. Biermann and J. Feldman. On the synthesis of finite-state machines from samples of their behavior. *Computers, IEEE Transactions on*, C-21(6):592–597, June 1972.

[9] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.*, 7(3):215–249, July 1998.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[11] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining behavior models from user-intensive web applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 277–287. ACM, 2014.

[12] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 591–600, New York, NY, USA, 2011. ACM.

[13] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: A survey. *SIGMOD Rec.*, 40(4):11–20, Jan. 2012.

[14] D. Lo and S.-C. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *13th Working Conference on Reverse Engineering (WCRE'06)*, pages 51–60. IEEE, 2006.

[15] D. Lo and 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*, pages 265–275. ACM, 2006.

[16] D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 345–354. ACM, 2009.

[17] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, pages 501–510. ACM, 2008.

[18] C. Luo, F. He, and C. Ghezzi. Inferring software behavioral models with mapreduce. In X. Li, Z. Liu, and W. Yi, editors, *Dependable Software Engineering: Theories, Tools, and Applications*, volume 9409 of *Lecture Notes in Computer Science*, pages 135–149. Springer International Publishing, 2015.

[19] A. V. Raman, J. D. Patrick, and 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 (ICML97)*, 1997.

[20] F. Thollard, P. Dupont, and C. d. l. Higuera. Probabilistic dfa inference using kullback-leibler

divergence and minimality. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 975–982, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[21] N. Walkinshaw and K. Bogdanov. Inferring finite-state models with temporal constraints. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 248–257. IEEE Computer Society, 2008.

[22] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Experience mining google's production console logs. In *Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, SLAML'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.