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), Beijing, China
luoc13@mails.tsinghua.edu.cn, hefei@tsinghua.edu.cn
[2] Key Laboratory for Information System Security, Ministry of Education, Beijing, China
[3] School of Software, Tsinghua University, Beijing 100084, China
[4] Politecnico di Milano, Milano, Italy
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 capacity of a single computer.

This paper focuses on inference of *behavioral models* and explores 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.

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 the software behavioral models automatically from execution logs [7,14].

Many model inference algorithms [4,10,13] 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 [18], which far exceeds the capacity of a single computer.

It is thus desirable to parallelize the processing of massive logs. In prior work [11], Lee et al. proposed an algorithm for slicing traces by parametric events. This algorithm is useful for log processing and model inference. However, one cannot parallelize this algorithm by simply dividing the trace into N segments and running N copies of the algorithm on these segments in parallel. Note that the events in different segments may be correlated and should be sliced together (Section 3). Processing the segments independently can lead to incorrect results.

To this end, we propose to use MapReduce [9] 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 these 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.

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 discusses the related work and Section 8 concludes this paper.

2 Approach Overview

2.1 MapReduce

MapReduce [9] 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 providing simple yet powerful interfaces to users. Hadoop [1] is an open-source implementation of MapReduce.

In MapReduce, the data is stored in a distributed file system (DFS) and the computation is based on key-value pairs. A MapReduce 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 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.

When solving a problem on top of MapReduce, one major concern is to design the distributed algorithm with 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 [12] for more information.

2.2 Behavioral Model Inference

The workflow of a typical behavioral model inference mainly consists of three steps: *log parsing*, *trace slicing*, and *model synthesis*. First, we rely on a parser to extract relevant events from the log files as defined in 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 then slices 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.

Consider the online shopping system shown in Figure 1 as a running example. The relevant events and their corresponding parameters are as follows:

- 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*,

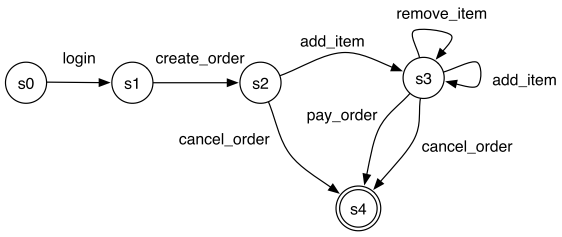
[1] http://hadoop.apache.org/

– the user *userid pays* the order *orderid*, and
– the user *userid cancels* the order *orderid.*

An example parametric trace excerpt for the system is shown in Figure 1a, and the behavioral model is depicted in Figure 1b.

```
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)
```

(a) A parametric trace



(b) The behavioral model

Fig. 1. An online shopping system example

2.3 Our Approach

To deal with the large logs generate by the software system, we propose to apply MapReduce to parallelize the model inference process. As shown in Figure 2, 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) [13]. The log parsing task is performed by mappers, while the trace slicing task is executed by reducers. 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.

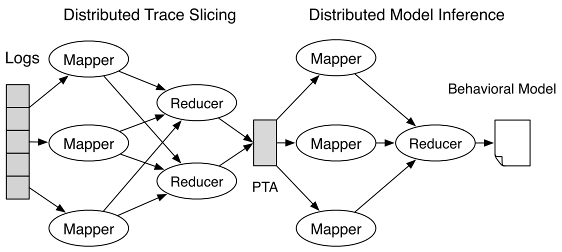


Fig. 2. Model inference with MapReduce

Although the basic algorithms for trace slicing [11] and model synthesis [7] 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 [11].

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 of interest and the parameters. The event specification for the running example 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 sets 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 the parameter names X and use the parameter values V_X to represent the parameter instance, if the mapping from X to V_X is clear from the context. For example, the parameter instance $\langle userid \mapsto user1, orderid \mapsto order1\rangle$ can be simplified 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 [11].

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 base event e (or parametric event $e\langle\theta\rangle$) appears in trace (or parametric trace) τ.*

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 $\theta'(x) = \theta(x)$.*

For example, $\langle user1\rangle$ is less informative than $\langle user1, order1\rangle$.

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 $\tau \upharpoonright_\theta$ 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 1a. For $\theta_1 = \langle user1, order1\rangle$, $\tau_1 \upharpoonright_{\theta_1}$ is the sequence of: *login, create_order, pay_order.*

A trace slice corresponds to a parameter instance. However, all parameter instances appearing in τ_1 are incomplete. With the following operator, 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:*

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

For example, the parameter instances $\langle user1, order1\rangle$ and $\langle order1, item1\rangle$ are compatible, and their combination gives $\langle user1, order1, item1\rangle$. 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. *Two parameter instances 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, θ_2 such that both θ_1 and θ_2 are τ-connected, $\theta_1 \bowtie \theta_2$, and $\theta = \theta_1 \sqcup \theta_2$.*

Considering the running example. The parameter instances $\langle user1, order1\rangle$ and $\langle order1, item1\rangle$ satisfy the first condition in 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 only consider trace slices for complete and connected parameter instances to avoid meaningless results as in [11].

4 Distributed Trace Slicing with MapReduce

This section presents our distributed trace slicing algorithm with MapReduce. The basic idea is to group all related parameter events and send then to the same reducer to generate correct trace slices. In the following, we first propose a data encoding mechanism, and then introduce the mapper and reducer functions.

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 determine to which reducer the parameter 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. Consider 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 1a is $\perp$. And the keys of the remaining parametric events in Figure 1a 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 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 parametric events labeled 1 and 3 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*, if* $key(e_1\langle\theta_1\rangle) \neq key(e_2\langle\theta_2\rangle)$*, then* $e_1\langle\theta_1\rangle$ *and* $e_2\langle\theta_2\rangle$ *must be* incompatible. [2]

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.

We now discuss how to choose $\mathcal{X}$ automatically. Since the 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.

[2] Due to space limitation, all proofs can be found in the extended version [15].

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, i.e., 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.

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 3 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 example trace in Figure 1a. Suppose there are two mappers and two reducers respectively. We assume each key-value pair output by the mappers is with the same label as the parametric event. Let $hash(\langle order1\rangle) = 1$ and $hash(\langle order2\rangle) = 2$. Then the key-value pairs labeled 1, 2, 3, 5, 8 are passed to $Reducer_1$; the key-value pairs labeled 1, 3, 4, 6, 7, 9 are passed to $Reducer_2$.

4.3 Reducer

Recall that during the shuffle phase, MapReduce merges 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 $values[]$.

The REDUCE function is shown in Figure 3. Note that all parametric events in $values[]$ are with the same key, but their parameter instances may be different. The RESTORE function first reorganizes $values[]$ into several lists (lines 3-6), each list $\Delta_{tmp}(\theta)$ corresponds to a parameter instance θ, and consists of base events

```
1: function MAP($line$)
2:     $e\langle\theta\rangle \leftarrow$ PARSE($line$);
3:     if $e\langle\theta\rangle = NULL$ then
4:         return ;
5:     if $\mathcal{X} \subseteq \mathcal{D}_e(e)$ then
6:         OUTPUT($\theta\restriction_{\mathcal{X}}, e\langle\theta\rangle$);
7:     else
8:         OUTPUT($\perp, e\langle\theta\rangle$);
```

```
1: function REDUCE($key$, $values$[])
2:     if $key = \perp$ then
3:         $\Delta_\perp \leftarrow$RESTORE($values$[]);
4:         return ;
5:     $\Delta \leftarrow$RESTORE($values$[]);
6:     while $\exists\theta_1 \in Dom(\Delta_\perp), \theta_2 \in Dom(\Delta)$
7: s.t. $\theta_1 \notin Dom(\Delta) \wedge \theta_1 \bowtie \theta_2$ do
8:         $\Delta(\theta_1) \leftarrow \Delta_\perp(\theta_1)$;
9:     CONSTRUCT($\Delta$);
```

```
1: function RESTORE($values$[])
2:     $\Delta_{tmp} \leftarrow \emptyset$;
3:     for $e\langle\theta\rangle \in values$[] do
4:         if $\theta \notin Dom(\Delta_{tmp})$ then
5:             Initialize $\Delta_{tmp}(\theta)$;
6:         Insert $e$ into $\Delta_{tmp}(\theta)$;
7:     return $\Delta_{tmp}$;
```

```
1: function CONSTRUCT($\Delta$)
2:     $\Omega \leftarrow Dom(\Delta)$;
3:     while $\exists\theta_1, \theta_2 \in \Omega$
4: s.t. $\theta_1 \bowtie \theta_2, (\theta_1 \sqcup \theta_2 \notin \Omega)$ do
5:         $\Omega \leftarrow \Omega \cup \{\theta_1 \sqcup \theta_2\}$;
6:     for complete $\theta \in \Omega$ do
7:         $\Gamma \leftarrow \{\Delta(\theta')|\theta' \sqsubseteq \theta, \theta' \in Dom(\Delta)\}$;
8:         $\tau\restriction_\theta \leftarrow$ merging event lists in $\Gamma$;
9:         Update $PTA$ using $\tau\restriction_\theta$;
```

Fig. 3. Distributed trace slicing

only. Here we abuse the notion of $Dom(\Delta)$, which denotes the set of parameter instances θ where the list $\Delta(\theta)$ is defined, i.e., $Dom(\Delta) = \{\theta|\Delta(\theta) \text{ is defined}\}$. Recall that each event is associated with a *timestamp*. At line 6, 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*.

Note that $\Delta_\perp$ is global and shared by multiple calls of the REDUCE function. And the MapReduce framework is configured such that key-value pairs in T_2 always come before pairs in T_1. 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 $\Delta_\perp(\theta_1)$ into Δ such that θ_1 can be combined with some $\theta_2 \in Dom(\Delta)$. According to Definition 9, if θ_1 and θ_2 are connected, and $\theta_1 \bowtie \theta_2$, then $\theta_1 \sqcup \theta_2$ is also *connected*, thus the list $\Delta_\perp(\theta_1)$ can be added to Δ (line 7). Note that θ_1 may again be strong compatible to other parameter instances in T_2; this process is thus iterative.

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 tries to combine (lines 3-5) all strong compatible parameter instances in Ω. This process is iterative, since the newly generated parameter instance may be combined to the existing ones. Then the trace slice for each complete and connected parameter instance θ is constructed by merging the event sequences of θ's less informative parameter instances (lines 7-9).

Consider $Reducer_1$ of our running example. After line 5 of the REDUCE function, $\Delta_\perp$ and Δ are defined as follows. For $\Delta_\perp$, $\Delta_\perp(\langle user1\rangle) = login$ and $\Delta_\perp(\langle user2\rangle) = login$. For Δ, $\Delta(\langle user1, order1\rangle) = create\_order, pay\_order$ and $\Delta(\langle order1, item1\rangle) = add\_item$. Then at line 6, since $\langle user1\rangle$ is strong compatible with $\langle user1, order1\rangle$, the list $\Delta_{\perp}(\langle user1\rangle)$ is added to Δ. After the while loop at line 3 of the CONSTRUCT function, $\Omega = \{\langle user1\rangle, \langle user1, order1\rangle, \langle order1, item1\rangle, \langle user1, order1, item1\rangle\}$. Let $\theta = \langle user1, order1, item1\rangle$, then $\tau\restriction_{\theta} = login, create\_order, add\_item, pay\_order$.

We take the prefix tree acceptor (PTA) as the intermediate structure. Each reducer keeps a partial PTA, which only maintains 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, and an example is shown in Figure 4.

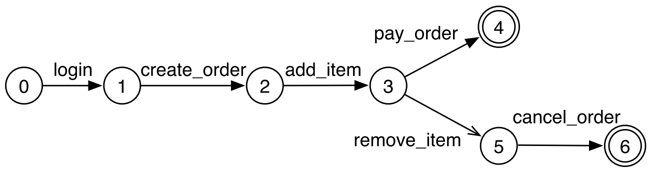


Fig. 4. PTA for the running example

5 Distributed Model Synthesis with MapReduce

Once the complete PTA has been generated, as previously shown, many off-the-shelf model synthesis algorithms [7,16] 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 further propose a distributed model synthesis algorithm based on k-tail [7] with MapReduce to improve efficiency.

The most expensive operation of k-tail 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 implement the distributed model synthesis algorithm with MapReduce, the intermediate results must 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.

We first introduce some notation relevant to the description 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 set of base events, S is a finite, non-empty set of states, $s_0 \in S$ is an initial state, σ is the state-transition function, and $F \subseteq S$ is the set of non-final states. Let $\sigma^* : S \times \Sigma^* \to S$ 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. 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^* [3].

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 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 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 $[s]$.

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

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 word ω in $\Sigma^{\leq k}$, $f(s,\omega) = 1$ 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 key of a given state s as the signature of s.

5.2 Mapper and Reducer

The pseudocode of distributed model synthesis is shown in Figure 5. 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 in M_{FSM} in correspondence to the given signature.

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 event e, a transition from $[s]$ to $[t]$ labeled e is added to M_{FSM}. The PostReduce function is called once and returns the synthesized model M_{FSM}.

[3] We do not require that a word ends in a final state, as in [8].

```
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 in $M_{FSM}$ w.r.t. $sig$;

6: function POSTREDUCE
7:     for each transition $(s_1, e, s_2)$ in $M_{PTA}$ do
8:         Add a transition ([$s_1$], $e$, [$s_2$]) in $M_{FSM}$;
```

Fig. 5. Distributed model synthesis

6 Experimental Evaluation

We implemented our approach on top of Hadoop 1.2.1, and conducted experiments on Amazon Elastic MapReduce clusters [4]. Each computing node 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.

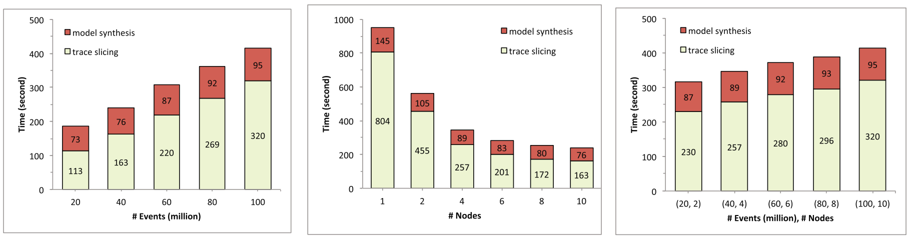
The datasets used in our experiments are synthetically generated as follows. (1) An automaton is randomly generated as the target model, which contains 50 states and maximally 5 transitions per state. (2) The automaton is randomly simulated to generate parametric traces. Each parametric trace is with 10 to 100 parametric events. (3) All generated parametric traces are randomly mixed up. (4) The same number of irrelevant entries are randomly added to the log as noises. 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. The size of the largest log file exceeds 10 GB.

We designed several sets of experiments to evaluate our approach, ranging from basic performance, speed up to scalability. The experimental results are reported and discussed below.

Basic Performance. The first set of experiments tests the running time of our approach for logs with increasing size. Sizes of these logs range from 20 to 100 million events. The cluster size is fixed to 10 nodes. The results are plotted in Figure 6a. 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 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 6b. We observed that the total running time

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



(a) Running time with inreasing log size

(b) Running time with increasing nodes

(c) Running time with increasing nodes and log size

Fig. 6. Experimental results

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 6c. 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.

Threat to Validity. The main threat to validity is the synthetic logs used in the evaluation. To mitigate this, the log generator is designed as practical as possible by imitating the practical parameter settings and the noises. Another possible threat to validity is certain characteristics of logs, e.g., the event definition, because of the heuristics we used for determining the parameter windows. To eliminate the bias involved in designing the data sets, we also choose synthetic logs and randomly generated event definitions in our evaluation.

7 Related Works

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

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. GK-Tail [14] extends the *k*-tail algorithm and infers extended finite state machines. Walkinshaw and Bogdanov [17] considered LTL constraints as additional input, and used model checking technique to guide the state merging process. Lo et al. [13] mined temporal invariants from execution traces and used the invariants to guide the model inference. Synoptic [5] adopted similar idea and incorporated refinement and coarsening to generate accurate but concise models. Lee et al. [11] proposed the trace slicing technique to mine parametric specifications. Ghezzi et al. [10] inferred users' behavior models from web application logs. However, to the best of our knowledge, there is no previously published work on applying MapReduce to model inference.

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 iterative algorithm for checking Linear Temporal Logic (LTL) formula over event traces with MapReduce. Bianculli et al. [6] further improved the work [2] by supporting 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. 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.

8 Conclusion

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

We plan to perform case studies on 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 [16] or incorporating temporal invariants [13] during inference phase.

Acknowledgment. This work was supported in part by the Chinese National 973 Plan (2010CB328003), the NSF of China (61272001, 91218302), the Chinese National Key Technology R&D Program (SQ2012BAJY4052), the Importation and Development of High-Caliber Talents Project of Beijing Municipal Institutions (YETP0167), and the Tsinghua University Initiative Scientific Research Program.

References

1. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2002, pp. 4–16. ACM, New York (2002)

2. Barre, B., Klein, M., Soucy-Boivin, M., Ollivier, P.-A., Hallé, S.: MapReduce for parallel trace validation of LTL properties. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 184–198. Springer, Heidelberg (2013)
3. Basin, D., Caronni, G., Ereth, S., Harvan, M., Klaedtke, F., Mantel, H.: Scalable offline monitoring. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 31–47. Springer, Heidelberg (2014)
4. Beschastnikh, I., Brun, Y., Ernst, M.D., Krishnamurthy, A.: Inferring models of concurrent systems from logs of their behavior with CSight. In: Proceedings of the 36th International Conference on Software Engineering, pp. 468–479. ACM (2014)
5. Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., Ernst, M.D.: 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, pp. 267–277. ACM (2011)
6. Bianculli, D., Ghezzi, C., Krstić, S.: Trace checking of metric temporal logic with aggregating modalities using MapReduce. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 144–158. Springer, Heidelberg (2014)
7. Biermann, A., Feldman, J.: On the synthesis of finite-state machines from samples of their behavior. Computers, IEEE Transactions on C **21**(6), 592–597 (1972)
8. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM Trans. Softw. Eng. Methodol. **7**(3), 215–249 (1998)
9. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
10. Ghezzi, C., Pezzè, M., Sama, M., Tamburrelli, G.: Mining behavior models from user-intensive web applications. In: Proceedings of the 36th International Conference on Software Engineering, pp. 277–287. ACM (2014)
11. Lee, C., Chen, F., Roşu, G.: Mining parametric specifications. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 591–600. ICSE 2011. ACM, New York (2011)
12. Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with mapreduce: A survey. SIGMOD Rec. **40**(4), 11–20 (2012)
13. Lo, D., Mariani, L., Pezzè, M.: 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, pp. 345–354. ACM (2009)
14. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proceedings of the 30th international conference on Software engineering, pp. 501–510. ACM (2008)
15. Luo, C., He, F., Ghezzi, C.: Inferring software behavioral models with mapreduce (extended version). http://sts.thss.tsinghua.edu.cn/beagle/paper/model-2015.pdf
16. Thollard, F., Dupont, P., Higuera, C.d.l.: Probabilistic dfa inference using kullback-leibler divergence and minimality. In: Proceedings of the Seventeenth International Conference on Machine Learning, ICML 2000, pp. 975–982. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2000)
17. Walkinshaw, N., Bogdanov, K.: Inferring finite-state models with temporal constraints. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 248–257. IEEE Computer Society (2008)
18. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.: Experience mining google's production console logs. In: Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques, SLAML 2010, pp. 5–5. USENIX Association, Berkeley, CA, USA (2010)