Scalable Parallelization of Specification Mining Using Distributed Computing

Shaowei Wang[1], David Lo[1], Lingxiao Jiang[1],

Shahar Maoz[2], and Aditya Budi[1]

[1]Singapore Management University, Singapore

[2]Tel Aviv University, Israel

{shaoweiwang.2010,davidlo,lxjiang,adityabudi}@smu.edu.sg

maoz@cs.tau.ac.il

Summary

Mining specifications from logs of execution traces has attracted much research effort in recent years since the mined specifications, either program invariants, temporal rules, association patterns, or various behavioral models, may be used to improve program documentation, comprehension, and verification. At the same time, a major challenge faced by most specification mining algorithms is related to their scalability, specifically when dealing with many large execution traces.

To address this challenge, we present a general, distributed specification mining algorithm that can parallelize and distribute repetitive specification mining tasks across multiple computers to achieve speedup proportional to the number of machines used. This general algorithm is designed based on our observation that most specification mining algorithms are data and memory intensive while computationally repetitive. To validate the general algorithm, we instantiate it with five existing sequential specification mining algorithms (CLIPPER, Daikon, k-tails, LM, and Perracotta) on a particular distributed computing model—MapReduce and one of its implementations Hadoop, to create five parallelized specification mining algorithms, and demonstrate the much improved scalability of the algorithms over many large traces ranging from 41MB to 157GB collected from seven DaCapo benchmark programs. Our evaluation shows that our parallelized Perracotta running on four machines (using up to eight CPU cores in total) speeds up the original sequential one by 3 to 18 times; The other four sequential algorithms are unable to complete analyzing the large traces, while our parallelized versions can complete and gain performance improvement by utilizing more machines and cores. We believe that our general, distributed algorithm fits many specification mining algorithms well, and can be instantiated with them to gain much performance and scalability improvements.

Keywords: Dynamic Analysis, Execution Profiles, Hadoop, MapReduce, Parallelization, Scalability, Specification Mining

Methods:

- Specification Mining Algorithms
    1. Frequent Pattern-Based Specification Miner
    2. Value-Based Invariant Miner
    3. Finite-State Machine Specification Miner
    4. Live Sequence Chart Miner
    5. Temporal Rule Miner
- Distributed Computing Models
    1. Message-Passing
    2. MapReduce
    3. Hadoop

1 Introduction

Specification mining is a family of program analysis techniques that extract likely specifications from code or execution traces. Specifications refer to certain patterns or properties that should hold in a program. They can take various forms, such as temporal rules about the order of certain method calls, and invariants that constrain method parameters and return values. The extracted specifications can provide much information about program properties which are not explicitly documented, and can be used to improve program documentation, comprehension, and verification tasks (Lo et al. 2011a).

An important challenge for many specification mining algorithms relates to their scalability since they need to take many potentially large program behavioral profiles as input to search for common patterns. A common way used to collect behavioral profiles is to execute a subject program with many test cases. In order to exercise the many behaviors of a large program, many test cases would need to be run. Also, the resultant execution traces are likely to be huge. The size of a code base, the number of test cases, the sizes of generated traces are all hurdles to the scalability of existing specification mining algorithms. For example, our evaluation on four existing specification mining algorithms, (1) CLIPPER (Lo et al. 2007a), a recurring pattern mining algorithm, (2) Daikon (Ernst et al. 2007), a value-based invariant mining algorithm, (3) k-tails (Kumar et al. 2011; Lorenzoli et al. 2008), a finite-state machine inference algorithm, and (4) LM (Lo et al. 2007b), a sequence diagram mining algorithm, shows that they fail to analyze the large traces ranging from 41MB to 157GB generated from seven DaCaPo benchmark programs (Blackburn et al. 2006). A fifth algorithm, Perracotta (Yang et al. 2006), a temporal rule mining algorithm, takes hours to analyze the traces before producing some specifications. In order to analyze many large traces from a large code base, existing specification mining algorithms need to be made much more scalable.

We observe that most specification mining algorithms are data intensive on one hand but computationally relatively repetitive on the other hand, and many repetitive tasks in those algorithms can be executed concurrently. Even though tasks in different algorithms may require various synchronization, the needed synchronization can be minimized with careful arrangements of the tasks to facilitate speedup when the algorithms are distributed onto multiple computers. This is the main insight that drives this chapter to address the scalability issue of many existing specification mining algorithms. Similar observations and ideas have been proposed to parallelize various algorithms in scientific computing, software engineering, data mining, and many other domains (e.g., (Dyer et al. 2013; Ene et al. 2011; MPIF 2012; Miliaraki et al. 2013; Shang et al. 2010)). However, as far as we know, there is little prior study on parallelization of various kinds of specification mining algorithms.

To help address the challenge of making various existing specification mining algorithms more scalable,[1] we propose a general specification mining algorithm that can perform repetitive specification mining tasks across multiple computers based on a general distributed computing model. The general algorithm is designed in such a way that itself abstracts away specific algorithmic details but captures the essences of many existing specification mining algorithms that mine specifications from program execution traces. We present this algorithm in the context of a message-passing based distributed computing model, in particular MapReduce. An algorithm designer can transform a *sequential* specification mining algorithm to a *distributed* one by following the guided steps in our algorithm and instantiating it with concrete algorithm-specific details.

[1]Note that it is not our goal to improve the accuracy of existing specification mining algorithms in inferring correct specifications. Rather, our goal is to improve their *scalability*. When evaluating the accuracy of the parallelized algorithms, we only need to compare it against their sequential versions, instead of human developers' ground truth.

To evaluate our general algorithm, we instantiate it with five existing sequential specification mining algorithms on top of a popular distributed computing model—MapReduce (MR) (Dean et al. 2004) and one of its open-source implementations—Hadoop (ASF 2013), and evaluate the scalability of the distributed versions of these algorithms. In particular, we show how we follow the guidance of our general algorithm, and use a common input-trace splitting scheme and several algorithm-specific techniques to divide and conquer five specification mining algorithms, (1) CLIPPER (Lo et al. 2007a), (2) Daikon (Ernst et al. 2007), (3) k-tails (Kumar et al. 2011; Lorenzoli et al. 2008), (4) LM (Lo et al. 2007b), and (5) Perracotta (Yang et al. 2006), and transform them into distributed ones. Note that the five algorithms produce different kinds of specifications expressed in different target languages, such as frequent patterns, value invariants, finite-state machines, sequence diagrams, and temporal rules. We evaluate the distributed algorithms on seven Java programs from the DaCapo benchmark (Blackburn et al. 2006) whose traces range from 41MB to 157GB. The results are encouraging. Perracotta's distributed version implemented within MapReduce (Perracotta^{MR}) running on four machines (using up to eight CPU cores in total) can speed up the original version by 3 to 18 times. The four other original algorithms are unable to analyze the large traces, while their distributed versions (CLIPPER^{MR}, Daikon^{MR}, k-tails^{MR}, and LM^{MR}) can complete within hours, and gain more performance improvement when more machines are employed.

Our main finding is that many specification mining algorithms fit distributed computing models well as they are comprised of many repetitive computational tasks dealing with data that may be split into partitions with limited overlapping. Our general algorithm also captures the essence of many specification mining algorithms well and can be used to help transform sequential algorithms into distributed ones to gain much performance and scalability improvements by implementing them within the MapReduce framework and executing them on clusters of computers. We believe our findings are applicable to many other specification mining algorithms, especially those that mine specifications expressed in one of the five target languages that we have investigated.

The contributions of this chapter are as follows:

1. Similar to many prior studies on parallelization of other algorithms in various domains, we observe that many *specification mining* algorithms can be fit into a distributed programming model, and much performance and scalability gains can be achieved by parallelizing them within a distributed computing framework, such as MapReduce.

2. We present a general distributed specification mining algorithm that abstracts away particular algorithmic details and represents the essences of many existing specification mining algorithms.

3. We propose an input-trace splitting scheme and several algorithm-specific techniques to instantiate the general algorithm with five existing sequential specification mining algorithms to create five distributed algorithms.

4. We perform an empirical evaluation with seven Java programs from the

DaCapo benchmark and show that the five distributed algorithms perform significantly faster than the original algorithms on many large traces.

This chapter is organized as follows. Section 2 provides a brief introduction to the five specification mining approaches and the distributed computing model we use in our work. Section 3 presents the main technical contribution of the chapter, that is, the general distributed specification mining algorithm and its instantiations with the five existing algorithms. Our implementation and empirical evaluation are described in Section 4. Section 5 discusses related work. Section 6 concludes with future work.

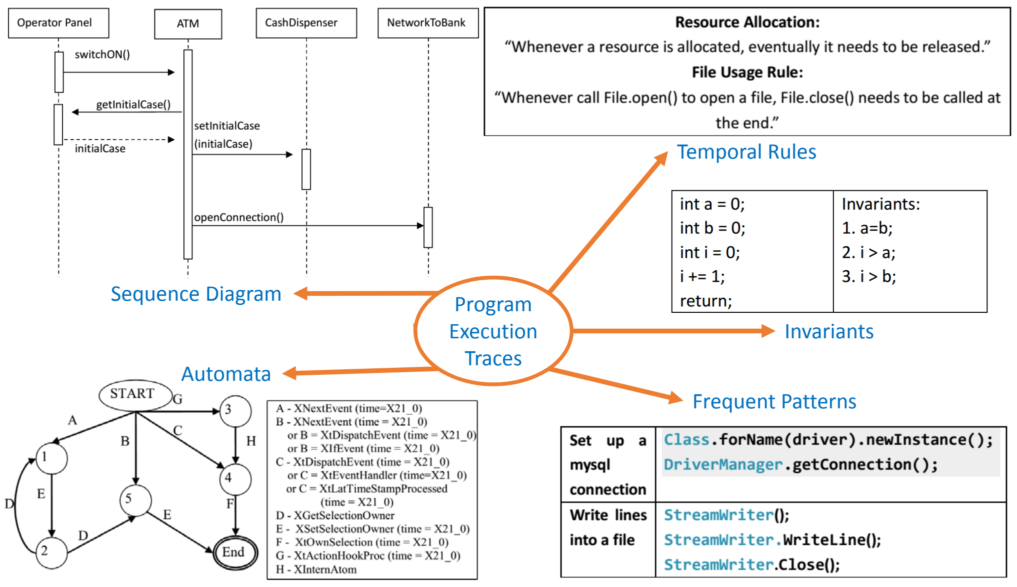
2 Background

In this section, we first briefly introduce each of the five mining algorithms that we parallelize. Then, we introduce the distributed computing model used in this chapter—the message-passing model and MapReduce.

2.1 Specification Mining Algorithms

Based on the format of the specifications that a specification mining algorithm produces (Lo et al. 2011a), many algorithms can be grouped into ones that produce (1) frequent patterns, (2) value-based invariants, (3) finite-state machines, (4) sequence diagrams, and (5) temporal rules. We briefly describe these families of algorithms in the following.

We present sample outputs of the five kinds specification mining algorithms in Figure 1, which can be mined from various kinds of program execution traces.



Resource Allocation:
"Whenever a resource is allocated, eventually it needs to be released."
File Usage Rule:
"Whenever call File.open() to open a file, File.close() needs to be called at the end."

int a = 0;
int b = 0;
int i = 0;
i += 1;
return;
	Invariants:
	1. a=b;
	2. i > a;
	3. i > b;

A - XNextEvent (time=X21_0)
B - XNextEvent (time = X21_0)
or B = XtDispatchEvent (time = X21_0)
or B = XIfEvent (time = X21_0)
C - XtDispatchEvent (time = X21_0)
or C = XtEventHandler (time=X21_0)
or C = XtLatTimeStampProcessed (time = X21_0)
D - XGetSelectionOwner
E - XSetSelectionOwner (time = X21_0)
F - XtOwnSelection (time = X21_0)
G - XtActionHookProc (time = X21_0)
H - XInternAtom

Set up a mysql connection	`Class.forName(driver).newInstance();`
	`DriverManager.getConnection();`

Write lines into a file	`StreamWriter();`
	`StreamWriter.WriteLine();`
	`StreamWriter.Close();`

Figure 1: Sample Outputs of Specification Mining Algorithms

2.1.1 Mining Frequent Patterns

Discovering patterns that appear many times in large input datasets is a well-known problem in data mining (Han et al. 2006). Many algorithms, such as frequent itemset mining, sequential pattern mining, and graph pattern mining, aim to capture frequent patterns. A number of algorithms specific to software engineering tasks have been proposed. For example, interaction pattern mining (El-Ramly et al. 2002) analyzes traces of system-user interactions to discover frequently recurring activities and uses them as parts of functional requirements for re-engineering. Iterative pattern mining (CLIPPER) (Lo et al. 2007a) takes in a set of execution profiles containing methods invoked during the executions and then identifies methods that often need to be invoked together or in a particular order as usage specifications for the methods.

2.1.2 Mining Value-Based Invariants

A value-based invariant captures the relation (e.g., `x==y`) among program variables that should be satisfied at a program point (e.g., when a method returns). Daikon is the pioneer and most well-known work extracting value-based invariants (Ernst et al. 2007). It has many invariant templates, such as Equality (e.g., `x==y`), IntGreaterThan (e.g., `iVal1>=iVal2`), IntArraySorted (e.g., `isSorted(iArray1)`), etc. Based on a set of input execution traces, Daikon matches the traces to the templates at various program points of interests (e.g., method entries and exits). Instances of the invariant templates satisfied by all (or most) of the input traces are outputted.

Value-based invariants generated by Daikon can be used independently, or be used in conjunction with other kinds of specifications, e.g., to enrich finite-state machines (Lorenzoli et al. 2008) or sequence diagrams (Lo et al. 2010).

2.1.3 Mining Finite-State Machines

Many of these algorithms extend or make use of techniques from the grammar inference community (Ammons et al. 2002; Biermann et al. 1972; Lorenzoli et al. 2008). One of these algorithms, k-tails (Biermann et al. 1972), builds a prefix tree acceptor from a set of execution traces that capture input-output behaviors; the nodes of the prefix tree acceptor are then merged based on some evaluation criteria, e.g., the similarity of the subsequent k-paths whose lengths are at most k, to form finite-state machines, which are then used as specifications of program behaviors.

2.1.4 Mining Sequence Diagrams

Sequence diagrams are a visual formalism to specify the ordering of events among components in a system. Different algorithms exist for mining various kinds of sequence diagrams, such as UML sequence diagrams (Briand et al. 2006), message sequence charts (Sousa et al. 2007), message sequence graphs (Kumar et al. 2011), live sequence charts (LSCs) (Damm et al. 2001; Harel et al. 2008; Lo et al. 2007b), etc. Such visual diagrams can help maintainers of a program to better understand how various components in the program interact with each other.

2.1.5 Mining Temporal Rules

Temporal rules can be expressed in various formats, such as association rules (Livshits et al. 2005; Yang et al. 2006), temporal logics (Lo et al. 2008b; Wasylkowski et al. 2011), "Whenever $x_1, \ldots, x_n$ occur, $y_1, \ldots, y_m$ also occur", etc. Such rules help to make it clearer which operations should or should not occur in certain orders so that maintainers of the program may make changes accordingly. Most temporal rule mining algorithms evaluate the validity of a rule based on the likelihood that the xs are followed by the ys, and the number of times x followed by y in the execution traces. They mainly differ in the semantics of the mined rules, the allowable values of n and m, and the metrics used to evaluate rule validity. For example, Perracotta extracts association rules of short length (n and m being 1) (Yang et al. 2006); Others extract temporal rules of longer lengths (Lo et al. 2008b).

2.2 Distributed Computing

Similar to many prior studies on parallelization of other algorithms in various domains, we observe that many specification mining algorithms can be broken into computational tasks that are repetitively applied to various parts of input data, and thus fit well to a distributed computing model. This section summarizes the concepts we need.

2.2.1 Message-Passing Model

We focus on distributed algorithms in the message-passing model where multiple processes on multiple computing nodes have their own local memory and communicate with each other by message passing, although our general algorithm may be adapted to other distributed computing models as well.

The processes share information by *sending/receiving* (or *dispatching/collecting*) data to/from each other. The processes most likely run the same programs, and the whole system should work correctly regardless of the messaging relations among the processes or the structure of the network. A popular standard and message-passing system is Message Passing Interface (MPI) defined in MPIF (2012). Such models themselves do not impose particular restrictions on the mechanism for messaging, and thus gives programmers much flexibility in algorithm/system designs. However, this also means that programmers need to deal with actual sending/receiving of messages, failure recovery, managing running processes, etc.

2.2.2 MapReduce

MapReduce is a simplified distributed computing model for processing large data in a cluster of computers (Dean et al. 2004), reducing programmers' burden of dealing with actual sending/receiving of messages and various system issues so that programmers may focus more on the algorithmic issues. It can be implemented on top of a message-passing system, such as MPI (Ho et al. 2010). In this chapter, we base our implementation on Hadoop (ASF 2013), a free and open-source implementation of MapReduce.

The model splits the problem at hand into smaller sub-problems as requested, distributes these sub-problems among the computers in the cluster, and collects and combines the results that are passed back. Besides the splitting function, the key to use the MapReduce framework is to define two functions: (1) *map*, which takes one input key/value pair (K_{ip}, V_{ip}), and generates zero or more intermediate key/value pairs (list(K_{int}, V_{int})), and (2) *reduce*, which composes all intermediate values associated with the same key into a final output. The splitting function, customizable to split input data differently, partitions the whole input dataset into small pieces, and transforms each piece into a set of key/value pairs.

Table 1: Map and Reduce Operations

Operation	**Input**	**Output**
map	(K_{ip},V_{ip})	list(K_{int},V_{int})
reduce	$(K_{int},\text{list}(V_{int}))$	O_{part}

MapReduce works by automatically applying the *map* function on each of the key/value pairs from the splitting function to produce an intermediate set of key/value pairs. It then automatically groups all intermediate values associated with the same key together; the *reduce* function is then applied to each group to result in a partial output O_{part}; all partial outputs are concatenated to form the final output. The inputs and outputs of the *map* and *reduce* functions are illustrated in Table 1. The following sections use the same symbols to represent the inputs and outputs of the functions as well.

3 Distributed Specification Mining

We now present the main contribution of this chapter: a general distributed specification mining algorithm and redefinitions of five concrete algorithms in the MapReduce model.

3.1 Principles

We first present our general principles and algorithm on parallelizing specification mining algorithms.

3.1.1 Abstracting Specification Mining Algorithms

Even though many specification mining algorithms are not initially designed to be distributed, we can *divide-and-conquer* by exploiting the parallelism in various parts of the algorithms based on our observation that many computational tasks in the algorithms are repetitively applied to various parts of input data. Fig. 2 illustrates the design idea of our general, distributed specification algorithm.

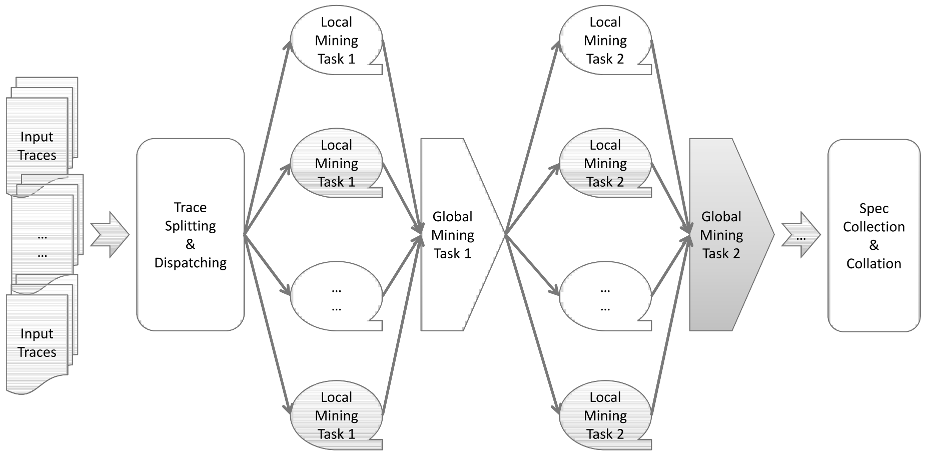


Figure 2: Overview of Our Distributed Specification Mining Algorithm

The key is to extract such tasks from existing algorithms that are repetitively applied to different parts of the input traces so that the input traces can be split and dispatched to and processed at different computing nodes. We note that many algorithms contain *local mining tasks* that can be done completely on a small part of the input data without the need of other parts. For some algorithms, however, there are still *global mining tasks* that need to operate on all data and we need to ensure those tasks can run scalably. Fortunately, we note that the mining algorithms rely on various "statistics" that measure the likelihood of a candidate specification to be valid. It is rarely necessary for the mining algorithms to really operate on all data at once in memory. Thus, we may also split the data (either the input traces or intermediate results from other tasks) needed by the global mining tasks so that they operate on smaller data and become more parallelizable and scalable; or, we can replace the global mining tasks with certain local ones plus certain specification compositions since many specifications are compositional. Multiple iterations of local and global mining tasks may be interweaved to find specifications minded by the normal sequential algorithms.

The general steps of our approach for parallelizing a given specification mining algorithm that takes a set of execution traces as input are as follows:

(1) Extract "local" operations in the algorithm that can be done in a separate trunk of the traces. The boundaries of trace trunks can be defined based on the operations and be algorithm-specific.

(2) Extract "global" operations in the algorithm that may need to be done with all data and decide how the data needed by the global operations may be split and/or replace the global operations with local ones.

(3) Split the input traces into trunks accordingly, and dispatch them to different computing nodes for either local or global operations.

(4) Collect results from different computing nodes, and compose them to

produce final specification outputs.

To produce efficient distributed versions of the sequential specification mining algorithms, one needs to ensure that the extracted local/global operations can be independent and executed concurrently with little or no synchronization. The steps described here are generic, although many details (what the local and global operations are, how to split data, how to dispatch/collect, how to compose results, etc.) are algorithm-specific, which are further explained in Section 3.2.

3.1.2 Distributed Specification Mining With MapReduce

MapReduce simplifies the general distributed computing model by providing automated mechanisms for setting up a "master" process that manages work allocated to "worker" processes, dispatching work to a worker, collecting results from a worker, recovering from failures, utilizing data locality, etc. We further describe our general specification mining steps in the context of MapReduce as follows:

(1) Define an appropriate *map* function that corresponds to a local mining task. Each instance of the *map* function runs in parallel with respect to other instances; it takes one trace trunk V_{ip} as input to produce a set of intermediate specification mining results (*intermediate key/value pairs*, list(K_{int},V_{int}), in MapReduce's terminology). The map function must be designed in such a way that the operation on V_{ip} is independent on other trace trunks.

(2) Define an appropriate *reduce* function that corresponds to a global mining

task or a composition operation that combines results (i.e., the intermediate key/value pairs, list(K_{int},V_{int})) from local mining tasks. We note that many algorithms rarely need global mining tasks and the composition operations may be as simple as concatenation or filtering or recursive applications of some local mining tasks (see algorithm-specific steps in Section 3.2).

(3) Define an appropriate *record reader* that splits input traces into trunks suitable for the map function. For example, if the map function from a mining algorithm deals with invariants within a method, a trace may be split at method entry and exit points. Each trace trunk can be identified by a *trace identifier* K_{ip} and its content V_{ip}.

(4) Let the MapReduce framework automatically handles the actual trace splitting, dispatching, and result collection.

We note that the above general steps provide guidance to make it easier to transform sequential specification mining algorithms into distributed ones, although strategies and techniques used to identify the local/global tasks in various algorithms might be different from each other, and there can be multiple ways to define the local/global operations, split the data, etc. for a given algorithm.

In the following, we describe our concrete instantiations of the general algorithm on five specification mining algorithms: (1) CLIPPER (Lo et al. 2007a), a recurring pattern mining algorithm, (2) Daikon (Ernst et al. 2007), a value-based invariant mining algorithm, (3) k-tails (Kumar et al. 2011; Lorenzoli et al. 2008), a finite-state machine inference algorithm, and (4) LM (Lo et al. 2007b), a sequence diagram mining algorithm, and (5) Perracotta (Yang et al. 2006), a temporal rule mining algorithm. We believe our findings can be easily adapted to other specification mining algorithms, especially those that mine specifications in languages similar to one of the five algorithms.

3.2 Algorithm-Specific Parallelization

3.2.1 Iterative Pattern Mining with MapReduce

We illustrate how to instantiate our general algorithm with CLIPPER, an iterative pattern mining algorithm (Lo et al. 2007a), to create the distributed version CLIPPER^{MR}.

CLIPPER, similar to many frequent pattern/sequence mining algorithms, explores the search space of all possible patterns. It starts with small patterns and then *grows* these patterns to larger ones. *Pattern growth* is performed repeatedly; each iteration grows a pattern by one unit. The iterations follow the depth-first search procedure. During the traversal of search space, every pattern that is *frequent* (i.e., appearing many times in the data set) is outputted. There have been studies on parallelization of frequent sequence mining algorithms, such as Miliaraki et al. (2013). Their work uses MapReduce too, but our data sources and the subject algorithms are specific to specification mining, which requires different parallelization strategies and techniques. The semantics of the sequential patterns mined by their approaches is also different from that of iterative patterns mined by CLIPPER. Their approach relies on w-equivalency

Algorithm 1 Generic Algorithm of Frequent Pattern Mining

```
 1: Procedure MinePatterns:
 2: Let SMALL = Small frequent patterns
 3: for all s in SMALL do
 4:     TraverseSSpace(s)
 5: end for
 6:
 7: Procedure TraverseSSpace(Pattern s):
 8: Output s
 9: Let NEXT= GrowPattern(s)
10: for all n in NEXT do
11:     TraverseSSpace(n)
12: end for
13:
14: Procedure GrowPattern(Pattern s):
15: Let BIGGER = s++e , where e is a growth unit and ++ is a grow operation
16: for all s' in BIGGER do
17:     if s' is frequent then
18:         Output s'
19:     end if
20: end for
```

property which may hold only for their sequential patterns.

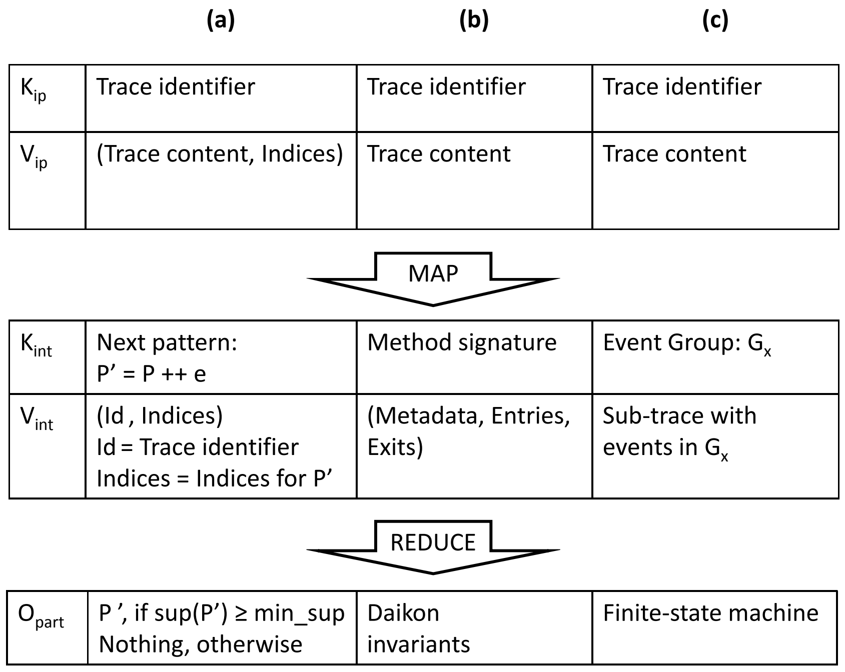
A piece of pseudocode for CLIPPER, as well as many frequent pattern mining algorithms, is shown in Algorithm 1. Intuitively, checking if a pattern is frequent or not could potentially be a parallelizable task. Unfortunately, it is not straightforward to break the pattern mining problem into *independent* tasks. On one hand, as we grow a pattern one unit at a time, if the pattern is not frequent, the longer pattern is not frequent either. In other words, some tasks can be omitted after the evaluation of other tasks and thus dependant on others. On the other hand, without the strategy of omitting longer patterns, the number of tasks grows exponentially with respect to the length of the traces.

Fortunately, we identify a common operation that is shared by these tasks, namely *pattern growth* (i.e., procedure `GrowPattern` in Algorithm 1). As pattern growth is performed many times, it is the critical operation that the mining algorithm spends much resource on. Thus, rather than trying to parallelize the whole pattern mining algorithm, we parallelize the pattern growth procedure.

The pattern growth procedure considers a pattern P and tries to extend it to patterns $P{+\!+}e$ where e is a growth unit and $+\!+$ is a growth operation (e.g., appending an event to an iterative pattern—from $\langle m1\rangle$ to $\langle m1, m2\rangle$).

For an iterative pattern P and trace T, we store the indices pointing to the various instances of the pattern in T. When we try to grow the pattern P to each pattern $P' \in \{P{+\!+}e\}$, we can update these indices to point to instances of the pattern P'. From the instances of P', we could then know if P' is frequent and thus should be in the output. Thus, we break this operation of checking *all* $P' \in \{P{+\!+}e\}$ into parallelizable tasks, each of which is in the following format: check if *one* pattern P' is frequent.

We realize `GrowPattern(Pattern P)` by instantiating the *map* and *reduce* functions in our general algorithm as follows. The *map* function works in



	(a)	(b)	(c)
K_{ip}	Trace identifier	Trace identifier	Trace identifier
V_{ip}	(Trace content, Indices)	Trace content	Trace content

MAP

	(a)	(b)	(c)
K_{int}	Next pattern: P' = P ++ e	Method signature	Event Group: G_x
V_{int}	(Id , Indices) Id = Trace identifier Indices = Indices for P'	(Metadata, Entries, Exits)	Sub-trace with events in G_x

REDUCE

	(a)	(b)	(c)
O_{part}	P ', if sup(P') ≥ min_sup Nothing, otherwise	Daikon invariants	Finite-state machine

Figure 3: MapReduce inputs (K_{ip}, V_{ip}), intermediate key/value pairs (K_{int}, V_{int}), and outputs (O_{part}) for GrowPattern(Pattern P) of CLIPPER (Column (a)), Daikon (Column (b)), and k-tails (Column (c))

parallel on each trace and updates the indices pointing to instances of P to indices of instances of $P' \in \{P{+}{+}e\}$. It creates an intermediate key/value pair (K_{int}/V_{int}) where the key corresponds to a P' and the value corresponds to the indices pointing to instances of P' in the trace. MapReduce groups all indices corresponding to a P'. Each intermediate key K_{int} and all of its corresponding intermediate values form a task that is sent to the *reduce* function. The *reduce* function computes the support of a pattern P' and outputs it if the support is more than the minimum support threshold min_sup (i.e., if P' is frequent).

We list the inputs (K_{ip}, V_{ip}), intermediate key/value pairs (K_{int}, V_{int}), and outputs (O_{part}) in the Column (a) in Fig. 3 for CLIPPER^{MR}.

Given a large execution trace, the pattern growth operation can be performed in parallel. Each trace is processed in parallel by multiple instances of the map function. Also, the process to check if a pattern P' is frequent or not could be done in parallel by multiple instances of the reduce function.

Note that we only parallelize the `GrowPattern` operation, and thus each MapReduce procedure in our implementation only performs one unit of pattern growth operation (i.e., $P \to P{+\!\!+}e$). Since many software properties are short and may be specified with only a few operation units (e.g., rules used in Static Driver Verifier (Microsoft)), we restrict the size of the patterns mined to be at most 3 to limit the experimental costs.

Example: Given two traces $trace_1 = \langle a, b, c\rangle$ and $trace_2 = \langle a, b\rangle$, we want to mine patterns with support values above $min\_sup = 2$ using $CLIPPER^{MR}$. In the first iteration, $CLIPPER^{MR}$ mines patterns of length 1. We have two instances of the map function map_1 and map_2 that take as input $trace_1$ and $trace_2$, respectively. Then map_1 creates the following intermediate key/value pairs: $\{K_{int} = \langle a\rangle, V_{int} = (trace_1, \{1\})\}$, $\{K_{int} = \langle b\rangle, V_{int} = (trace_1, \{2\})\}$, and $\{K_{int} = \langle c\rangle, V_{int} = (trace_1, \{3\})\}$. Map function map_2 produces $\{K_{int} = \langle a\rangle, V_{int} = (trace_2, \{1\})\}$, and $\{K_{int} = \langle b\rangle, V_{int} = (trace_2, \{2\})\}$. We have three instances of the reduce function: $reduce_1$, $reduce_2$, and $reduce_3$. $reduce_1$ takes pairs with the key $\langle a\rangle$ and checks whether the number of instances is larger than min_sup. Similarly, $reduce_2$ and $reduce_3$ collects pairs with the keys $\langle b\rangle$ and

$\langle c \rangle$, respectively. The reduce functions output patterns $\langle a \rangle$ and $\langle b \rangle$ because they satisfy the threshold min_sup. In the next iteration, $CLIPPER^{MR}$ mines patterns of length 2. The *map* functions generate the following intermediate key/value pairs: $\{K_{int} = \langle a, b \rangle, V_{int} = (trace_1, \{1\})\}$, $\{K_{int} = \langle a, c \rangle, V_{int} = (trace_1, \{1\})\}$, $\{K_{int} = \langle b, c \rangle, V_{int} = (trace_1, \{2\})\}$, and $\{K_{int} = \langle a, b \rangle, V_{int} = (trace_2, \{1\})\}$. The *reduce* functions group the instances based on the key values and find that pattern $\langle a, b \rangle$ satisfies the min_sup threshold. Finally, $CLIPPER^{MR}$ would return the following frequent patterns: $\langle a \rangle$, $\langle b \rangle$, and $\langle a, b \rangle$.

3.2.2 Value-Based Invariants Mining with MapReduce

We parallelize Daikon into a distributed version Daikon^{MR} by instantiating our general algorithm in MapReduce.

Similar to Daikon, Daikon^{MR} takes as input a set of execution traces and outputs invariants for each method that hold for all execution traces. We parallelize Daikon by splitting the input traces: Rather than feeding the whole set of traces to one instance of Daikon, we process the traces for each method separately, and in parallel—the trace logs for each method are fed into one instance of Daikon. This allows us to instantiate our general algorithm for Daikon relatively easily without the need for synchronization because the traces of different methods are independent from one another for inferring method-level invariants.

In Daikon^{MR}, the *map* function processes a set of traces and outputs $\langle method\ signature, (metadata, entries, and\ exits) \rangle$ pairs. The latter part of each pair contains method metadata (e.g., the number of parameters a method has, the types of the parameters, etc.), and parts of the execution traces corresponding to the states of the various variables when entries and exits of the methods are executed. The *reduce* function runs an instance of Daikon on $(metadata, entries, and\ exits)$ of the same method and outputs $\langle method\ signature, method\ invariants \rangle$ pair. We illustrate the inputs, intermediate key/value pairs, and outputs for Daikon^{MR} in MapReduce's terminology in the Column (b) in Fig. 3.

Many instances of Daikon are executed in parallel, each of which runs on a rather small input. Thus, each instance of Daikon requires much less memory and is able to quickly produce a subset of the results.

3.2.3 Finite-State Machine Mining with MapReduce

Many finite-state machine mining algorithms are variants of the k-tails algorithm (Biermann et al. 1972). The algorithms investigate a set of execution traces and produce a single finite-state machine. However, this finite-state machine may be too large and difficult to comprehend. Many studies propose methods to *split* the traces and learn a finite-state machine for each sub-trace whose ideas can be instantiated in MapReduce with our general algorithm, and we name it as k-tails^{MR}.

Consider a mapping function: EVENTS→GROUP, where EVENTS is the set of events (i.e., method calls) in the execution traces, and GROUP is a group identifier. Events in the same group are *related*. Many notions of relatedness may be defined (c.f., (Pradel et al. 2009)). In this chapter we consider one such notion: a group of events is composed of invocations of methods appearing in the same class.

The *map* function slices each trace into a set of sub-traces based on the group membership of each event. MapReduce collects the sub-traces belonging to the same group. The *reduce* function produces one finite-state machine for a group of sub-traces by invoking an instance of the k-tails algorithm. We illustrate the inputs, intermediate key/value pairs, and outputs for k-tails^{MR} in the Column (c) in Fig. 3.

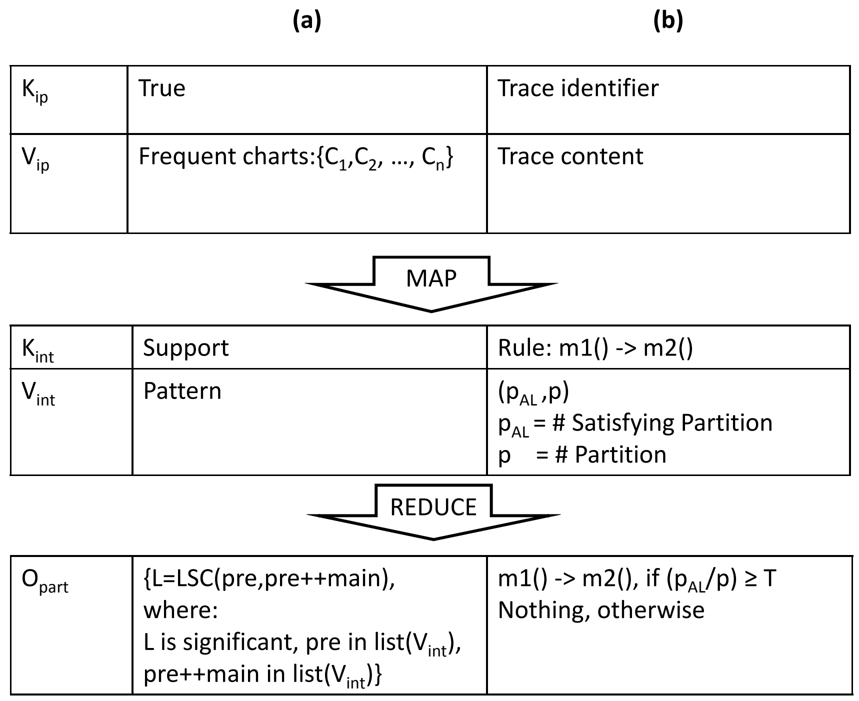
The slicing could be done in parallel for separate execution traces. In addition, the learning of multiple finite-state machines could be done in parallel.

3.2.4 Sequence Diagram Mining with MapReduce

We illustrate how to transform the LM algorithm (Lo et al. 2007b) that mines live sequence charts (LSC) (Damm et al. 2001; Harel et al. 2008) into LM^{MR}.

An LSC contains two parts: pre-chart and main-chart. The semantics of LSC dictates that whenever the pre-chart is observed, eventually the main-chart would also be observed. The goal of the mining task is to find all LSCs that appear frequently (i.e., the support of the LSC is more than min_sup), and that the proportion of pre-chart being followed by main-chart in the execution traces is greater than a certain min_conf threshold. LSCs obeying these criteria are considered significant.

The LSC mining algorithm works in two steps:



	(a)	(b)
K_{ip}	True	Trace identifier
V_{ip}	Frequent charts:{C_1,C_2, …, C_n}	Trace content

K_{int}	Support	Rule: m1() -> m2()

V_{int}	Pattern	(p_{AL} ,p)
		p_{AL} = # Satisfying Partition
		p = # Partition

O_{part}	{L=LSC(pre,pre++main), where: L is significant, pre in list(V_{int}), pre++main in list(V_{int})}	m1() -> m2(), if (p_{AL}/p) ≥ T Nothing, otherwise

Figure 4: MapReduce inputs (K_{ip}, V_{ip}), intermediate key/value pairs (K_{int}, V_{int}), and outputs (O_{part}) for LM (Column (a)), and Perracotta (Column (b))

(1) Mine frequent charts;

(2) Compose frequent charts into significant LSCs.

For the first step, we employ the same strategy as described in Section 3.2.1. For the second step, we consider the special case of min_conf=100% (i.e., mining of LSCs that are always observed in the execution traces—the pre-chart is always eventually followed by the main-chart).

In LSC mining, Lo and Maoz define *positive witnesses* of a chart C, denoted by $pos(C)$, as the number of trace segments that obey chart C (Lo et al. 2008a).

They also define *weak negative witnesses* of C, denoted by $w\_neg(C)$, as the number of trace segments that do not obey C due to end of trace being reached. The *support* of an LSC $L = pre \rightarrow main$ is simply the number of positive witnesses of $pre{+\!\!+}main$. The *confidence* of an LSC L is given by:

$$conf(L) = \frac{|pos(pre{+\!\!+}main)|+|w\_neg(pre{+\!\!+}main)|}{|pos(pre)|}$$

We first note that LSCs with 100% confidence must be composed of pre-chart and main-chart where $|pos(pre{+\!\!+}main)| + |w\_neg(pre{+\!\!+}main)|$ equals $|pos(pre)|$. We could break the task of constructing all significant LSCs to sub-tasks: find all significant LSCs of a particular support value.

We name the MapReduce version of LM as LM^{MR}. For LM^{MR}, we use the following *map* and *reduce* functions. The *map* function works on the set of patterns and simply group pattern C, where either $pos(C) + w\_neg(C)$ or $pos(C)$ has a particular value into a bucket. If a pattern C has different $pos(C)+ w\_neg(C)$ and $pos(C)$ values, it is put into two buckets. The *reduce* function constructs significant LSCs by composing two patterns in each bucket. We list the involved inputs, intermediate key/value pairs, and outputs in the Column (a) in Fig. 4.

The composition of charts into LSCs is done in parallel for separate buckets. If the significant LSCs have many different support values, the speed up in the second stage of LSC mining due to the parallelization could be substantial.

Finally, note that LM^{MR} applies MapReduce framework twice, sequentially composed in a pipeline. The first application computes the frequent charts using the solution presented in Section 3.2.1. The output of this application is used as an input for the second application described above. We note that composition of instances of MapReduce in a pipeline is also common (e.g., (Chambers et al. 2010)).

3.2.5 Temporal Rule Mining with MapReduce

We illustrate how to re-implement basic Perracotta (Yang et al. 2006), by using our general algorithm and MapReduce. Perracotta proposes several extensions and variants to the main algorithm, e.g., chaining, etc. We consider only the basic Perracotta, which computes alternating properties. We call the resultant algorithm Perracotta^{MR}.

For n unique methods in the execution traces, Perracotta checks n^2 possible temporal specifications of the format: "Whenever method m_1 is executed, eventually method m_2 is executed" (denoted as $m_1 \rightarrow m_2$), to see if the specification is strongly observed in the execution traces. A measure known as *satisfaction rate* is defined based on the proportion of partitions in the traces that satisfy $m_1^+ m_2^+$ (i.e., p) that also satisfy the temporal rule $m_1 \rightarrow m_2$ (i.e., p_{AL}). It is often the case that n is large and Perracotta would take a lot of memory to process the traces together. We break the original task into small subtasks by splitting each long trace into smaller sub-traces of size k each and process them independently—by default we set k to be 300,000 events for Perracotta^{MR}. As k is relatively large, by the principle of locality (i.e., related events appear close together; c.f., (Gabel et al. 2010)), there will be no or little loss in the mined specifications.

Following our general algorithm, we define the following *map* and *reduce* functions. The *map* function is applied to each execution subtrace independently. For each execution subtrace, the *map* function computes for each potential rule $m_i \rightarrow m_j$ two numbers: the number of partitions in the subtrace (i.e., p) and the number of partitions in the subtrace that satisfy the rule (i.e., p_{AL}). The method pair is the intermediate key, while the two numbers p and p_{AL} are the value in the intermediate key/value pair (K_{int} and V_{int}). MapReduce groups the counts for the same rule together. The *reduce* function simply sums up the p and p_{AL} for the separate execution sub-traces and computes a *satisfaction rate* for the corresponding rule. Rules that satisfy a user-defined threshold of satisfaction rate (i.e., S), are provided as output. By default, the satisfaction rate is 0.8. We list the involved inputs, intermediate key/value pairs, and outputs in the Column (b) in Fig. 4.

Notice that the sub-traces can now be processed in parallel using multiple runs of the *map* and *reduce* functions on potentially different machines. Also, the computation and checking of the satisfaction rate could be done in parallel. No synchronization is needed among different sub-traces.

4 Implementation & Empirical Evaluation

We have implemented the algorithms described in the previous sections in Hadoop (ASF 2013), one of the most popular MapReduce implementations.

We describe our datasets, experimental settings, research questions, and experimental results in the following.

4.1 Dataset & Experimental Settings

We use seven programs, avrora, batik, fop, luindex, lusearch, xalan, and tomcat, from the DaCapo benchmark (Blackburn et al. 2006) as our subjects. We also have implemented a Java instrumentation tool to collect all methods that get executed (referred to as *trace databases* later) for the experiments with CLIPPER, k-tails, LM, and Perracotta; we use Chicory, a part of Daikon, to collect the traces for Daikon. The sizes for Daikon traces for these seven programs range from 18GB to 157GB, while the sizes of the trace databases range from 41MB to 533MB. The experiments are run on four Acer M680G machines, each having an Intel Core i5 4-core CPU, 4GB of memory, and 2TB of hard disk, installed with the operating system Ubuntu version 12.04. One of the machines is used as the master; three others are slaves. We also configure Hadoop (version 2.0.0-alpha) to use *up to* three cores for distributed map and reduce tasks on each slave machine to reduce the effects of potential resource contentions. We set the maximum memory of each map/reduce task to 1,200 MB and leave the other settings to their default values (e.g., the Hadoop file system's replication factor). Before running the MR versions of the specification mining algorithms, we also copy all traces from the usual ext4 file system under Ubuntu into the Hadoop file system as a one-time cost. To reduce experimental bias, we run each experiment with each version of the various specification mining algorithms two times and report the averages across the two runs.

4.2 Research Questions & Results

Our study aims to answer the following research questions:

RQ1 Could existing specification mining algorithms scale to process large execution traces?

RQ2 Could MapReduce be used to improve the scalability of existing specification mining algorithms?

RQ3 How much more scalable our mining algorithms would be when we increase the number of processing cores?

We discuss the answers to the above research questions for each of the five specification mining algorithms.

4.2.1 Mining Frequent Patterns

To answer RQ1, we run the original version of CLIPPER on the traces. This version mines patterns recursively and needs to load the complete trace database into memory. Thus even for the smallest trace database with size of 41MB (from batik), original CLIPPER is unable to run.

To answer RQ2, we examine the performance of CLIPPER^{MR} with up to eight parallel map and reduce tasks. CLIPPER$^{MR(8)}$ (that is, with eight parallel map and reduce tasks) outputs the invariants for all traces from the seven programs within 1493 minutes. This shows CLIPPER^{MR} improves the scalability of the original CLIPPER.

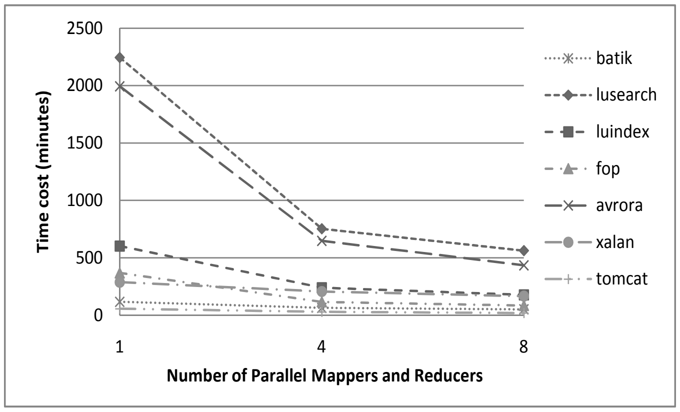


Figure 5: Performance Improvements for CLIPPER

To answer RQ3, we compare the time cost for CLIPPER^{MR} as we increase the number of parallel tasks (see Fig. 5). We find that the performance improves as we increase the number of parallel tasks. By increasing the number of parallel MapReduce tasks from one to four, we gain a speed up in range of 1.4 to 3.2 times. By increasing the number of parallel MapReduce tasks from one to eight, we gain a speed up in range of 1.7 to 4.6 times. The reason why CLIPPER^{MR} cannot speed up as much as parallelized versions of the other mining algorithms (see below) is that it needs to process a lot of I/O operations across different nodes in Hadoop system.

4.2.2 Mining Value-Based Invariants

To answer RQ1, we run the original Daikon on the traces. Since the traces from the seven programs are all larger than 18GB, the original Daikon runs out of memory before outputting any invariant.

To answer RQ2, we examine the performance of the original Daikon with that of Daikon^{MR} with up to eight parallel map and reduce tasks. Daikon$^{MR(8)}$ outputs the invariants for *all* traces from seven programs within 2374 minutes, and we are only unable to infer the invariants of less than 5% of the methods (since we terminate a Daikon instance if it takes more than 600 seconds to complete). Obviously, Daikon^{MR} improves the scalability of the original Daikon.

To answer RQ3, we compare the time cost for Daikon^{MR} as we increase the number of parallel tasks (see Fig. 6). We find that the performance improves when we increase the number of parallel tasks. By increasing the number of parallel MapReduce tasks from one to four, we gain a speed up in range of 2.7 to 3 times. By increasing the number of parallel MapReduce tasks from one to eight, we gain a speed up in range of 4.2 to 5.4 times. We notice that the rate of speed up decreases as we increase the number of parallel tasks from 4 to 8. This is so as there are more resource contentions between the mappers and reducers as the number of parallel tasks are increased in our small four-machine cluster with limited memory.

4.2.3 Mining Finite-State Machines

To answer RQ1 and RQ2, we compare the performance of the original k-tails with that of k-tails$^{MR(8)}$. The original k-tails ran out of memory before outputting any finite-state machines. On the other hand, k-tails$^{MR(8)}$ is able to output finite-state machines for *all* programs in 40 minutes. Similar to Daikon^{MR}, we also employ a timeout and terminate an instance of k-tails construction process run in one reducer if it does not complete within 120

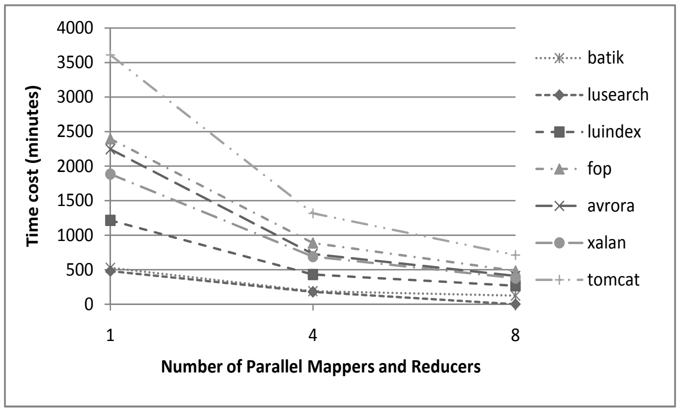


Figure 6: Performance Improvements for Daikon

seconds. We find that we are unable to run k-tails to completion for only 5% of the classes. Obviously, k-tails^{MR} improves the scalability of the original k-tails.

To answer RQ3, we compare the time cost for k-tails^{MR} as we increase the number of parallel tasks (see Fig. 7). We find that the performance improves as we increase the number of parallel tasks. By increasing the number of parallel MapReduce tasks from one to four, we gain a speed up in range of 2 to 3.7 times. By increasing the number of parallel MapReduce tasks from one to eight, we gain a speed up in range of 2.3 to 5.6 times.

4.2.4 Mining Sequence Diagrams

To answer RQ1 and RQ2, we compare the performance of the original LM with that of LM$^{MR(8)}$. Original LM is unable to run because of memory problems, while LM^{MR} is able to get the sequence diagrams (since LM^{MR} is based on CLIPPER^{MR}). LM$^{MR(8)}$ can output the invariants for all traces from the seven programs within 1508 minutes. This shows LM^{MR} can improve the scalability

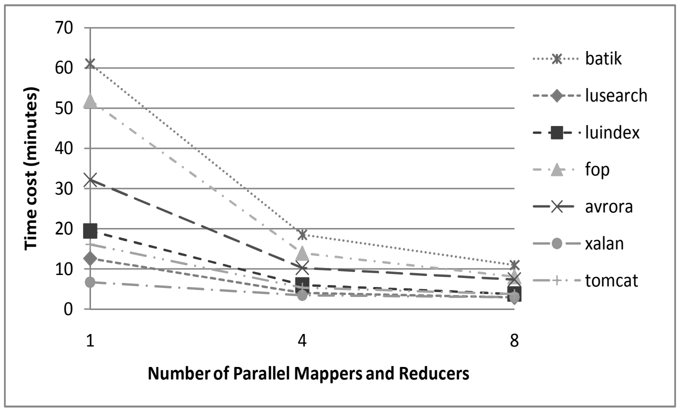


Figure 7: Performance Improvements for k-tails

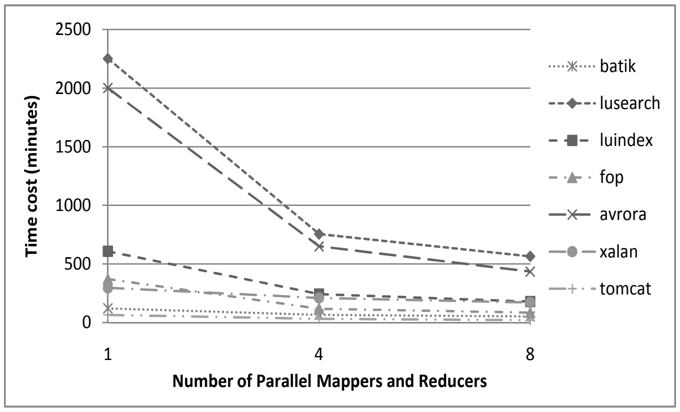


Figure 8: Performance Improvements for LM

of original LM.

To answer RQ3, we compare the time cost for LM^{MR} as we increase the number of parallel tasks (see Fig. 8). We find that the performance improves as we increase the number of parallel tasks. By increasing the number of parallel MapReduce tasks from one to four, we gain a speed up in range of 1.4 to 3 times. By increasing the number of parallel MapReduce tasks from one to eight, we gain a speed up in range of 1.7 to 4.6 times. The performance improvement of LM^{MR} over LM is similar with that of CLIPPER^{MR} over CLIPPER. This is reasonable as the first of the two steps of LM^{MR} is based on CLIPPER^{MR} (see Section 3.2.4), and the second step for composing frequent charts into significant LSCs takes little time with respect to the time spent in the first step.

4.2.5 Mining Temporal Rules

To answer RQ1, we mine temporal rules with the original Perracotta, which was able to mine the temporal rules from all of the traces. Perracotta's memory cost is quadratic to the number of unique events in the traces. In our study, the unique events are the methods that get invoked when program is run. The number of unique events is not so big, which is no more than 3000.

To answer RQ2, we compare the performance of the original Perracotta with that of Perracotta^{MR} and present the results in Figure 9. We see that Perracotta$^{MR(8)}$ achieves speed up in range of 3.5 to 18.2. Note that Perracotta$^{MR(8)}$ may achieve more than 8 times speed up than Perracotta. This can be related to the fact that Perracotta is a memory-intensive algorithm (with a space complexity $O(n^2)$ and a time complexity $O(nL)$, where n is the number of unique events in the traces and L is the total length of all traces). Its sequential version needs to load all traces into memory sequentially, while the parallelized version may load many split smaller traces into memory simultaneously even when there is only one core available for map/reduce tasks. At the same time, the accuracy of Perracotta^{MR} is 100% with respect to Perracotta: when

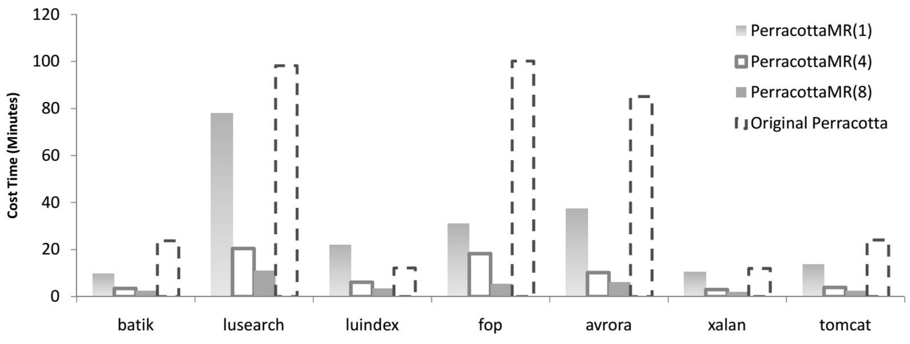


Figure 9: Original versus Parallelized Perracotta

we compare the output of Perracotta^{MR} with that of Perracotta, there is no temporal rule that is missed.

To answer RQ3, we compare the time cost for Perracotta^{MR} as we increase the number of parallel tasks (see Fig. 10). We find that the performance improves as we increase the number of parallel tasks. By increasing the number of parallel MapReduce tasks from one to four, we gain speed up in range of 1.6 to 3.8 times. By increasing the number of parallel map reduce tasks from one to eight, we gain speed up in range of 4.1 to 7 times. Figure 10 also shows that the rate of speed up decreases as we increase the number of parallel tasks from 4 to 8. This is so as there are more resource contentions between the mappers and reducers as the number of parallel tasks are increased in our small four-machine cluster.

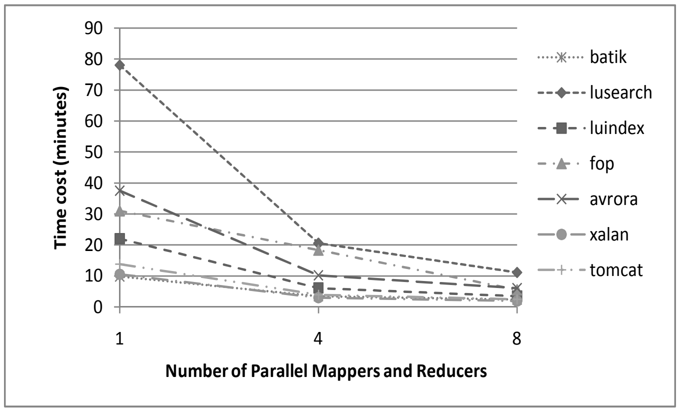


Figure 10: Performance Improvements for Perracotta

4.3 Threats to Validity and Current Limitations

In this work, we have considered five families of specification mining algorithms: those mining frequent patterns, value-based invariants, finite-state machines, sequence diagrams, and temporal rules. For each family, we have considered one algorithm. There are other specification mining algorithms that we do not consider in this study, e.g., those that analyze program code rather than execution traces (Li et al. 2005; Nguyen et al. 2009; Shoham et al. 2007; Wasylkowski et al. 2011; Weimer et al. 2005). It is not clear if our approach can be easily extended to all other specification mining algorithms. In this study, we modify and adapt the algorithms to follow a divide-and-conquer strategy; it is not clear if all specification mining algorithms can be modified to follow this strategy. In the future, we would like to investigate even more algorithms and algorithm families and show how they can be modified to follow appropriate divide-and-conquer strategies to leverage the power of MapReduce.

We have evaluated our approach on seven programs from the DaCapo benchmark (Blackburn et al. 2006). This benchmark has been used extensively in many past studies, e.g., (Bond et al. 2010; Chen et al. 2007). Still, these programs might not be representative of all open-source and industrial software systems. We plan to reduce this threat to validity further by investigating more programs in addition to those in the DaCapo benchmark in the future. Also, we have experimented with a cluster of four machines running eight cores. In the future, we plan to extend our experiment with even more machines and more cores. However, even with four machines, we have shown how the power of MapReduce could be tapped to scale various specification mining algorithms.

One limitation is imposed by the implementation of MapReduce that we use, i.e., Hadoop. One of the most important issues for a distributed platform is locality, as network bandwidth is the bottleneck when processing a large amount of data. To solve this problem, Hadoop attempts to replicate the data across the nodes and to always locate the nearest replica of the data. However, a substantial proportion of the time may still be spent on data transmission, typically for algorithms which involve heavy data transmission load. Our experiments use the default Hadoop file system replication factor 3 (i.e., each block of data is replicated to 3 machines) to minimize the transmission overheads during computation. The speedup factor of the parallelized versions of the specification mining algorithms may be affected if more machines are used or the replication factor is changed. In the future, we plan to perform a more comprehensive investigation of the effect of data transmission load, identify factors that may significantly affect the performance of distributed specification mining algorithms, and design improved algorithms that can reduce such overheads further.

5 Related Work

We now discuss closely related studies on specification mining, uses of MapReduce in software engineering, and parallelizing data mining algorithms in general. This section is by no means a complete list of related work.

5.1 Specification Mining and Its Applications

Mined specifications can help developers to understand legacy systems (Yang et al. 2006), to detect potential bugs (Wasylkowski et al. 2011). They can also be used as input for model checkers and for the purpose of program verification (Li et al. 2010) or be converted into test cases (Dallmeier et al. 2010).

Some families of specification mining algorithms have been described in Section 2. Here we describe other related recent work. Beschastnikh et al. (2011) mines three kinds of temporal invariants from system logs and merges them into a state-based behavioral model. Wu et al. (2011) proposes an approach that mines specifications from a variety of API data including information from API client programs, library source code, and comments. Lo et al. (2011b) uses three concepts, equivalence classes among LSCs, isomorphic embeddings, and delta-discriminative similarity measures, to mine a succinct set of LSCs and improve the readability of mining results. Alrajeh et al. (2012)

presents a semi-automated approach that detects vacuously satisfiable scenarios leveraging a model checker and generates new scenarios to avoid the vacuity using machine learning (inductive logic programming). Kumar et al. (2011) presents a framework for mining message sequence graphs that can represent concurrent behaviors of a distributed system. Zhong et al. (2009) infers API specifications from documentations expressed in English. Lee et al. (2011) implements a tool jMiner to mine parametric specifications by using a concept of trace slicing. The proposed approach first slices independent interactions from program traces. The independent interactions are then fed into to a variant of the k-tails algorithm to produce a probabilistic finite-state machine. Wei et al. (2011) builds on top of Daikon to infer contracts for Eiffel programs.

All five specification mining algorithms that we study in this chapter (CLIPPER, Daikon, k-tails, LM, and Perracotta) analyze program execution traces. Other approaches to specification mining use program code as input (e.g., (Li et al. 2005; Nguyen et al. 2009; Shoham et al. 2007; Wasylkowski et al. 2011; Weimer et al. 2005)). A technique called *DySy*, mines invariants, similar to those generated by Daikon, by performing symbolic execution to reduce the number of test cases needed to mine the invariants and improve the quality of the mined invariants (Csallner et al. 2008). There are also algorithms that mine specifications in forms different from the five families described in Section 2, e.g, algebraic specifications (Henkel et al. 2008) and separation logic invariants (Magill et al. 2006). In the future, it would be interesting to examine how one can apply our general algorithm and MapReduce to the above diverse mining algorithms.

Some recent studies propose to better understand, extend, and compare existing specification mining algorithms. An approach called *InvariMint*, allows users to construct a model inference algorithm by using a declarative specification (Beschastnikh et al. 2013). They show that their approach could help users understand, extend, and compare algorithms that mine specifications in the form of finite-state machines. Different from their work, we propose an approach to adapt existing algorithms to the MapReduce framework and make them more scalable and efficient.

A comprehensive survey of past studies on specification mining is available from a recent journal paper (Robillard et al. 2013) and a book on specification mining (Lo et al. 2011a).

5.2 MapReduce in Software Engineering

Shang et al. (2009, 2010) have presented experience reports on scaling tools for mining software repositories (MSR) using MapReduce. They investigate several case studies to analyze the potential of the MapReduce platform to scale up MSR tools, including (1) J-REX, which mines a CVS repository for calculating changes of software metrics over the history of a software project, (2) CC-Finder, which is a token-based clone detection tool designed to extract code clones from systems developed in several programming languages, and (3) JACK, which is a log analyzer that uses data mining techniques to process system execution logs and automatically identify problems in load tests. Specification mining approaches are not covered in these studies. Recently, Dyer et al. (2013) propose a language and an infrastructure called Boa to ease the analysis of software repositories. Users can specify queries in a domain specific language and these queries can be processed by Boa's processing engine which uses the MapReduce distributed computing model.

Different from these studies, we specifically focus on specification mining algorithms and investigate the potential of using MapReduce to make them more scalable.

5.3 Parallel Data Mining Algorithms

Kang et al. (2011) uses MapReduce to propagate beliefs on a sparse billion-node graph. Liu et al. (2009) uses MapReduce to parallelize an algorithm inferring document relevance for web search. Ene et al. (2011) speeds up general clustering algorithms by using MapReduce. Miliaraki et al. (2013) recently proposes a parallelization of a frequent sequence mining algorithm that can run on MapReduce. Their approach relies on w-equivalency property that holds only for sequential patterns, and the semantics of the sequential patterns mined by their approach is different from that of iterative patterns mined by $CLIPPER$ (which is the closest algorithm, considered in this chapter, to the frequent sequence mining algorithm). Although our approach employs MapReduce too, our data sources and the subject algorithms are specific to specification mining, which requires different parallelization strategies and techniques.

6 Conclusion and Future Work

In this chapter, we address the challenge of making specification mining algorithms scalable. We have presented a general algorithm design that helps to transform sequential specification mining algorithms into distributed ones based on the observation that many specification mining algorithms are data-intensive but computationally repetitive. In particular, we have shown how five different kinds of algorithms, CLIPPER, Daikon, k-tails, LM, and Perracotta, can be parallelized by following our general algorithm and leveraging a popular distributed computing framework MapReduce. We have evaluated the distributed versions of these algorithms with seven programs from the DaCapo benchmark and find that the distributed versions can significantly improve the scalability of the original algorithms for every trace dataset of sizes ranging from 41MB to 157GB. The distributed Perracotta running on four machines (using up to eight CPU cores in total) speeds up the original version by 3 to 18 times. The original CLIPPER, Daikon, k-tails, and LM are unable to handle the large traces, while our distributed versions can complete in hours and much performance improvements can be gained by utilizing more machines.

We consider the following for future work. First, our distributed algorithms are not necessarily optimal; we plan to investigate whether defining the map and reduce functions differently and/or splitting input data differently would improve the scalability of these algorithms. For example, Daikon has more than one hundred invariant templates that are checked against traces when it looks for actual invariants. Checking each kind of template is independent from others, and thus could be parallelized as a map function as well. Second, our distributed algorithms are only evaluated with gigabyte traces in a four-machine cluster with many default settings; we would like to evaluate them with terabyte traces in a commercial cluster and see how performance improves when the number of processors increases and various cluster system settings are used. Third, we consider the application of our general algorithm and MapReduce to additional kinds of specification mining algorithms not covered in this chapter, e.g., algorithms leveraging other information besides traces (e.g., text, software repository). Fourth, some variants of the algorithms we have investigated may also deserve special attention, e.g., the variants of LSC mining triggers and effects (Lo et al. 2008a), or the combination of scenario-based and value-based invariants (Lo et al. 2010).

References

Alrajeh, Dalal, Jeff Kramer, Alessandra Russo, and Sebastián Uchitel (2012). “Learning from Vacuously Satisfiable Scenario-Based Specifications”. In: *FASE*, pp. 377–393.

Ammons, Glenn, Rastislav Bodík, and James R. Larus (2002). “Mining specifications”. In: *POPL*, pp. 4–16.

Apache Software Foundation (2013). *Hadoop*. URL: `http://hadoop.apache.org/`.

Beschastnikh, Ivan, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst (2011). “Leveraging existing instrumentation to automatically infer invariant-constrained models”. In: *SIGSOFT FSE*, pp. 267–277.

Beschastnikh, Ivan, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy (2013). “Unifying FSM-inference algorithms through declarative specification”. In: *ICSE*, pp. 252–261.

Biermann, A.W. and J.A. Feldman (1972). “On the synthesis of finite-state machines from samples of their behavior”. In: *IEEE Transactions on Computers* 21, pp. 591–597.

Blackburn, Stephen M. et al. (2006). “The DaCapo benchmarks: Java benchmarking development and analysis”. In: *OOPSLA*, pp. 169–190.

Bond, Michael D., Katherine E. Coons, and Kathryn S. McKinley (2010). “PACER: proportional detection of data races”. In: *PLDI*, pp. 255–268.

Briand, Lionel C., Yvan Labiche, and Johanne Leduc (2006). “Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software”. In: *IEEE Trans. Software Eng.* 32.9, pp. 642–663.

Chambers, Craig, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum (2010). “FlumeJava: Easy, efficient data-parallel pipelines”. In: *PLDI*, pp. 363–375.

Chen, Feng and Grigore Rosu (2007). “Mop: an efficient and generic runtime verification framework”. In: *OOPSLA*, pp. 569–588.

Csallner, Christoph, Nikolai Tillmann, and Yannis Smaragdakis (2008). “DySy: Dynamic symbolic execution for invariant inference”. In: *ICSE*, pp. 281–290.

Dallmeier, Valentin, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller (2010). “Generating test cases for specification mining”. In: *ISSTA*, pp. 85–96.

Damm, Werner and David Harel (2001). “LSCs: Breathing Life into Message Sequence Charts”. In: *Formal Methods in System Design*, pp. 45–80.

Dean, J. and S. Ghemawat (2004). “MapReduce: Simplified Data Processing on Large Clusters”. In: *OSDI*, pp. 107–113.

Dyer, Robert, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen (2013). “Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories”. In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE ’13. San Francisco, CA, USA, pp. 422–431. ISBN: 978-1-4673-3076-3.

El-Ramly, Mohammad, Eleni Stroulia, and Paul G. Sorenson (2002). “From runtime behavior to usage scenarios: an interaction-pattern mining approach”. In: *KDD*, pp. 315–324.

Ene, Alina, Sungjin Im, and Benjamin Moseley (2011). “Fast clustering using MapReduce”. In: *KDD*, pp. 681–689.

Ernst, Michael D., Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao (2007). “The Daikon system for dynamic detection of likely invariants”. In: *Sci. Comput. Program.* 69.1-3, pp. 35–45.

Gabel, Mark and Zhendong Su (2010). “Online inference and enforcement of temporal properties”. In: *ICSE*, pp. 15–24.

Han, J. and M. Kamber (2006). *Data Mining: Concepts and Techniques*. Morgan Kauffman.

Harel, David and Shahar Maoz (2008). “Assert and negate revisited: Modal semantics for UML sequence diagrams”. In: *Software and Systems Modeling* 7.2, pp. 237–252.

Henkel, Johannes, Christoph Reichenbach, and Amer Diwan (2008). “Developing and debugging algebraic specifications for Java classes”. In: *ACM TOSEM* 17.3, 14:1–14:37.

Ho, Yu-Fan, Sih-Wei Chen, Chang-Yi Chen, Yung-Ching Hsu, and Pangfeng Liu (2010). “A Mapreduce programming framework using message passing”. In: *International Computer Symposium (ICS)*, pp. 883–888.

Kang, U, Duen Horng Chau, and Christos Faloutsos (2011). “Mining large graphs: Algorithms, inference, and discoveries”. In: *ICDE*, pp. 243–254.

Kumar, Sandeep, Siau-Cheng Khoo, Abhik Roychoudhury, and David Lo (2011). “Mining message sequence graphs”. In: *ICSE*, pp. 91–100.

Lee, Choonghwan, Feng Chen, and Grigore Rosu (2011). “Mining parametric specifications”. In: *ICSE*, pp. 591–600.

Li, Wenchao, Alessandro Forin, and Sanjit A. Seshia (2010). “Scalable specification mining for verification and diagnosis”. In: *DAC*, pp. 755–760.

Li, Zhenmin and Yuanyuan Zhou (2005). “PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code”. In: *ESEC/SIGSOFT FSE*, pp. 306–315.

Liu, Chao, Fan Guo, and Christos Faloutsos (2009). "BBM: bayesian browsing model from petabyte-scale data". In: *KDD*, pp. 537–546.

Livshits, V. Benjamin and Thomas Zimmermann (2005). "DynaMine: finding common error patterns by mining software revision histories". In: *ESEC/SIGSOFT FSE*, pp. 296–305.

Lo, David, Siau-Cheng Khoo, and Chao Liu (2007a). "Efficient mining of iterative patterns for software specification discovery". In: *KDD*, pp. 460–469.

Lo, David, Shahar Maoz, and Siau-Cheng Khoo (2007b). "Mining modal scenario-based specifications from execution traces of reactive systems". In: *ASE*, pp. 465–468.

Lo, David and Shahar Maoz (2008a). "Mining Scenario-Based Triggers and Effects". In: *ASE*, pp. 109–118.

Lo, David, Siau-Cheng Khoo, and Chao Liu (2008b). "Mining temporal rules for software maintenance". In: *Journal of Software Maintenance* 20.4, pp. 227–247.

Lo, David and Shahar Maoz (2010). "Scenario-based and value-based specification mining: better together". In: *ASE*, pp. 387–396.

Lo, David, Siau-Cheng Khoo, Jiawei Han, and Chao Liu, eds. (2011a). *Mining Software Specifications: Methodologies and Applications*. CRC Press Data Mining and Knowledge Discovery Series.

Lo, David and Shahar Maoz (2011b). "Towards Succinctness in Mining Scenario-Based Specifications". In: *ICECCS*, pp. 231–240.

Lorenzoli, Davide, Leonardo Mariani, and Mauro Pezzè (2008). “Automatic generation of software behavioral models”. In: *ICSE*, pp. 501–510.

Magill, Stephen, Aleksandar Nanevski, Edmund Clarke, and Peter Lee (2006). “Inferring Invariants in Separation Logic for Imperative List-processing Programs”. In: *SPACE*.

Message Passing Interface Forum (2012). *MPI: A Message-Passing Interface Standard*. URL: `http://www.mpi-forum.org/docs/docs.html`.

Microsoft. *Static Driver Verifier: DDI Compliance Rules*. URL: `http://msdn.microsoft.com/en-us/library/ff552840(v=vs.85).aspx`.

Miliaraki, Iris, Klaus Berberich, Rainer Gemulla, and Spyros Zoupanos (2013). “Mind the gap: large-scale frequent sequence mining”. In: *SIGMOD Conference*, pp. 797–808.

Nguyen, Tung Thanh, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen (2009). “Graph-based mining of multiple object usage patterns”. In: *ESEC/SIGSOFT FSE*, pp. 383–392.

Pradel, Michael and Thomas R. Gross (2009). “Automatic Generation of Object Usage Specifications from Large Method Traces”. In: *ASE*, pp. 371–382.

Robillard, Martin P., Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford (2013). “Automated API Property Inference Techniques”. In: *IEEE Trans. Software Eng.* Pp. 613–637.

Shang, Weiyi, Zhen Ming Jiang, Bram Adams, and Ahmed E. Hassan (2009). “MapReduce as a general framework to support research in Mining Software Repositories (MSR)”. In: *MSR*, pp. 21–30.

Shang, Weiyi, Bram Adams, and Ahmed E. Hassan (2010). "An experience report on scaling tools for mining software repositories using MapReduce". In: *ASE*, pp. 275–284.

Shoham, Sharon, Eran Yahav, Stephen Fink, and Marco Pistoia (2007). "Static specification mining using automata-based abstractions". In: *ISSTA*, pp. 174–184.

Sousa, Felipe Cantal de, Nabor C. Mendonça, Sebastián Uchitel, and Jeff Kramer (2007). "Detecting Implied Scenarios from Execution Traces". In: *WCRE*, pp. 50–59.

Wasylkowski, Andrzej and Andreas Zeller (2011). "Mining temporal specifications from object usage". In: *Autom. Softw. Eng.* 18.3-4, pp. 263–292.

Wei, Yi, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer (2011). "Inferring better contracts". In: *ICSE*, pp. 191–200.

Weimer, Westley and George C. Necula (2005). "Mining Temporal Specifications for Error Detection". In: *TACAS*, pp. 461–476.

Wu, Qian, Guang-Tai Liang, Qian-Xiang Wang, and Hong Mei (2011). "Mining effective temporal specifications from heterogeneous API data". In: *J. Comput. Sci. Technol.* 26.6, pp. 1061–1075.

Yang, Jinlin, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das (2006). "Perracotta: mining temporal API rules from imperfect traces". In: *ICSE*, pp. 282–291.

Zhong, Hao, Lu Zhang, Tao Xie, and Hong Mei (2009). “Inferring Resource Specifications from Natural Language API Documentation”. In: *ASE*, pp. 307–318.