J-ReCoVer: Java Reducer Commutativity Verifier*

Yu-Fang Chen[1,2], Chang-Yi Chiang[2], Lukáš Holík[3], Wei-Tsung Kao[1], Hsin-Hung Lin[1], Tomáš Vojnar[3], Yean-Fu Wen[2], and Wei-Cheng Wu[1]

[1] Institute of Information Science, Academia Sinica, Taiwan
[2] Graduate Institute of Information Management, National Taipei University, Taiwan
[3] FIT, IT4I Centre of Excellence,Brno University of Technology, Czechia

Abstract. The MapReduce framework for data-parallel computation was first proposed by Google [10] and later implemented in the Apache Hadoop project. Under the MapReduce framework, a reducer computes output values from a sequence of input values transmitted over the network. Due to non-determinism in data transmission, the order in which input values arrive at the reducer is not fixed. In relation to this, the *commutativity problem* of reducers asks if the output of a reducer is independent of the order of its inputs. Indeed, there are several advantages for a reducer to be commutative, e.g., the verification problem of a MapReduce program can be reduced to the problem of verifying a sequential program. We present the tool J-ReCoVer (Java Reducer Commutativity Verifier) that implements effective heuristics for reducer commutativity analysis. J-ReCoVer is the first tool that is specialized in checking reducer commutativity. Our experimental results over 118 benchmark examples collected from open repositories are very positive; J-ReCoVer correctly handles over 97 % of them.

1 Introduction

MapReduce belongs among the most popular frameworks for data parallel computation. A MapReduce program [10] consists of several pairs of *mappers* and *reducers* running on a machine cluster for handling big data in parallel. Usually, mappers and reducers are the only components in a MapReduce program that involve concurrency. Mappers read data from a distributed database and output a sequence of *key-value* pairs. The elements of the sequence (i.e., key-value pairs) with the same key are sent to the same reducer for further processing. Due to scheduling policies and network latency, the same inputs may arrive at a reducer in different orders in different executions. Therefore, reducers are typically required to be *commutative*, that is, the output of a reducer is required to be independent of the order of its inputs. The problem of checking whether this is indeed the case is known as the *commutativity problem* of reducers [9, 17, 6, 8].

If a reducer is commutative, it will have the same external behaviour under all possible schedules, and one then suffices with considering any chosen interleaving of input values when examining its behaviour instead of having to consider all of them. By fixing a schedule, the verification problem of a MapReduce program reduces to the verification problem of a sequential program, which is known to be much easier than the verification problem of concurrent programs.

* The work was supported by the Minister of Science and Technology of Taiwan project 106-2221-E-001-009-MY3 and the Czech Science Foundation project 17-12465S.

On the other hand, the non-commutative behaviour of a reducer is often the source of very tricky bugs. A study conducted by Microsoft investigated the commutativity problem of 508 reducers running on their MapReduce server [17]. These reducers were carefully checked using all traditional means such as code review, testing, and experiments with real data for months. Still, five of these programs contained very subtle bugs caused by non-commutativity (which was confirmed by the programmers).

However, checking reducer commutativity is a difficult problem on its own right [6, 8, 7]. Even for a simple case in which all values are mathematical integers, it is proved undecidable in [6]. For the case when all values are machine integers (e.g., 64-bits integers), the problem is decidable, but the only available algorithm, which was proposed in [6] too, is of very high complexity and hence of theoretical interest only.

In this paper, we present the J-ReCoVer tool (Java Reducer Commutativity Verifier), which is available at `http://www.jrecover.tk/`. The tool implements a heuristic approach for checking the commutativity problem that—despite its simplicity—works very efficiently on a large set of practical integer reducer programs as shown by our experiments. The main ingredient of the approach is a reduction from the commutativity problem to an SMT problem. The reduction is incomplete but sound. It is accompanied with several heuristics which enable the approach to scale to real-world examples. For the case when the reducer is not proven commutative, we complement the approach by using testing to find concrete counterexamples.

We collected benchmarks from open repositories such as GitHub and Bitbucket to evaluate J-ReCoVer. With the help of a search engine *searchcode.com* over those repositories, we collected 118 programs. We provide this collection of programs to other interested researchers as a side contribution of the paper. Our tool J-ReCoVer is able to correctly analyse all but three of the programs.

Related Work. The reducer commutativity problem can be reduced to a *program equivalence* problem. One creates another program R' that first non-deterministically swaps two consecutive input values and then executes the code of R. If R' and R are equivalent, using the fact that all permutations of a list can be obtained by swapping consecutive list elements finitely many times[4], R can be proved to be commutative. A series of research works address program equivalence checking (or closely related topics such as regression verification and translation validation), cf. [15, 11, 3, 13] to name a few.

From a high-level view, checking equivalence of two programs P and P' can be reduced to a sequential verification problem by executing P' after P, followed by checking whether the two programs always produce the same outputs. The approach can be made more efficient by finding the right *synchronization points* and combining the code of P and P' in an interleaved manner. A lot of research effort have been invested into finding good synchronization points. In this work, we propose the *head of the top-level reducer loop* as the synchronization point suitable for reducer commutativity analysis. According to our experience, discussed later on, the reducers usually contain just a single such loop. Moreover, for the case when there are more top-level loops in a reducer, we propose a way of breaking the reducer into several ones to be checked independently.

[4] Here is an example to produce [3;2;5;1;4] from [1;2;3;4;5] by swapping consecutive elements: [**1**;**2**;3;4;5] → [2;**1**;**3**;4;5] → [2;3;1;**4**;**5**]→ [2;3;**1**;**5**;4]→ [**2**;**3**;5;1;4]→ [3;2;5;1;4].

However, we observe that if one naively reduces the commutativity problem to an equivalence problem and checks it in a precise manner, many reducers cannot be verified. Therefore, J-ReCoVer uses an over-approximation of the reducer's behaviour. This approximation allows for a much more efficient, yet—according to our experiments—precise enough commutativity analysis.

Our approach can be seen as using some form of *sequentialisation* of the concurrent behaviour. Sequentialisation is the key approach behind many current successful approaches for verifying multithreaded programs [14, 12]. However, our sequentialisation approach is specialised for the case of reducers and quite different from what is used in sequentialisation of multithreaded programs: indeed, in MapReduce programs there is no notion of threads nor context switches.

Various forms of sequentialisation are also used in works dealing with the concept of *robustness* of event-driven asynchronous programs [5] or works dealing with programs running under some *relaxed memory models* [4, 2, 1]. However, their computation models are again quite different from that of reducers, and their results cannot be directly applied. Besides verification, another interesting research direction, using commutativity analysis as a component, is *synthesis* of MapReduce programs [16].

2 Notations and Definitions

We use $[n,m]$ to denote the set of integers $\{k \mid n \leq k \leq m\}$ and lift the equality predicate $=$ to tuples in the standard, component-wise, way.

To present our approach, we introduce a highly simplified language for describing reducers. Let `Var` be a set of *integer variables*. An *integer expression* in `Exp` can either be a variable from `Var`, a constant value, a call to the $cur()$ function that reads and consumes an input value of the reducer, a non-deterministically chosen integer value $*$, or a combination of integer expressions over basic arithmetic operations. A *command* in `Cmd` can be an assignment, a branch statement, a sequence of commands, or an $out(v)$ statement that outputs the value of $v \in$ `Var`. A *reducer program* is defined as $s_1;\mathbf{Loop}\{s_2\};s_3$ where $s_1, s_2, s_3 \in$ `Cmd`. According to our observation over hundreds of reducer programs in open repositories, reducer programs are almost always in this form. The $\mathbf{Loop}\{s_2\}$ statement enters the loop body to execute s_2 repeatedly for each input element until the entire input list is consumed. An example of a reducer is shown in Figure 1. In the paper, to simplify the presentation, we assume that a reducer does never produce any output in the loop body s_2. J-ReCoVer implements an algorithm to deal with an output inside the loop (as briefly mentioned at the end of Section 4.2).

$s := 0; c := 0;$
Loop{
  $s := s + cur();$
  $c := c + 1$
}
$; o := s/c;$
$out(o)$

Fig. 1: A reducer that computes the average value.

Some reducers use two (or more) top-level loops to compute the output, possibly interleaved with some non-looping code. These loops are executed sequentially, repeatedly iterating over the input list from its beginning. For example, for calculating the standard deviation, one first computes the average of inputs and then uses it to compute the final result (using two passes over the input list). In that case, we suggest to verify

the reducer by first partitioning it into two (or more) reducers, each containing a single top-level loop, and then verifying these reducers separately.[5] The top-level loops communicate through shared variables. After the transformation, reducers corresponding to the second top-level loop (and possibly further such loops) will work with random initial values of the shared variable, which over-approximates the original behaviour. In our experience, the second (or further) top-level loop are usually commutative even with arbitrary initial shared variable values, and so J-ReCoVer can be used to handle such reducers.

3 Overview of the J-ReCoVer Tool

The input of J-ReCoVer is a reducer program written in Java, which is the most popular programming language used in the Hadoop MapReduce framework. The J-ReCoVer tool has three main components, *Preprocessor*, *Prover*, and *BugFinder*. As the name suggests, Preprocessor reads as input a reducer program and performs the required preprocessing. The goal of Prover is to show that a given reducer is commutative, and the goal of BugFinder is the opposite. The architecture of J-ReCoVer can be found in Figure 2. The user can input a reducer program to J-ReCoVer either through our web-interface or use a binary application installed on his/her own machine.

The *Preprocessor component* first compiles a reducer program to bytecode and uses the tool Soot[6] to further convert it to the so-called "Jimple" format, which is an intermediate language designed to simplify the analysis of Java programs. Under the Hadoop MapReduce framework, the permutation of the input is handled by the scheduler/shuffler component and is affected by issues like network latency, which are not controllable by programmers. In order to deal with such issues, we wrote our own dummy Hadoop environment for the reducer as a part of the Preprocessor component so that the input order of the reducer is now controlled by J-ReCoVer. Finally, the Preprocessor performs a program transformation to simplify the analysis.

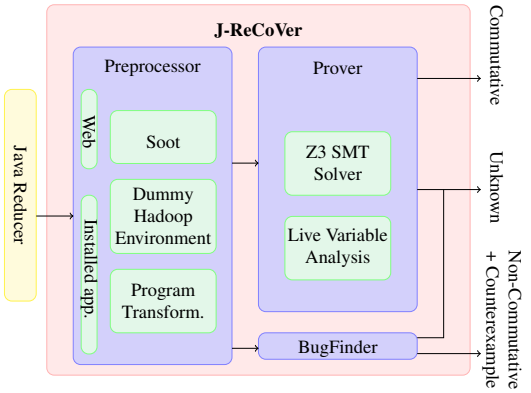

Fig. 2: Overview of the J-ReCoVer tool.

The *BugFinder component* generates random pairs of lists, with the list of each pair being permutations of each other. A concrete counterexample is reported if the reducer outputs different results for the two lists of a generated pair. Our procedure for

[5] Nested loops can be removed by adding additional branch statements since both the inner and outer loop are over the same input list. In fact, such a program construct has never occurred in the examples we have seen.

[6] https://github.com/Sable/soot

generating random pairs is quite naive. We use five different input list of lengths 5, 7, 9, 11, and 13. For each length, we generate 100 lists and pick uniformly at random one of its permutations. Although the approach is simple; in practice, it finds counterexamples in all of our non-commutative benchmarks in few seconds.

The *Prover component* reduces the commutativity problem to an SMT problem. From a high-level point of view, we are checking equivalence between a reducer program and its variant that has two consecutive inputs swapped. We show that this equivalence check can be reduced to a first-order formula and give it to the SMT solver Z3 for solving. In case that Z3 proves the formula unsatisfiable, we know that swapping any two consecutive inputs of the reducer will not change its output. Since all permutations of a list can be obtained by swapping consecutive elements finitely many times, it follows that the reducer outputs the same value for all permutations of the same list of inputs. In this case, J-ReCoVer stops and reports that the reducer is commutative.

4 The Preprocessor and the Prover

Before entering the Prover component, the Preprocessor first performs a program transformation to simplify the verification task (Section 4.1). The output of the preprocessor is a commutativity-equivalent reducer program. The algorithm of the Prover is then explained in Section 4.2.

4.1 The Program Transformation in the Preprocessor

In real-world reducers, it is often the case that the s_1 part of the reducer reads from the input. Since our reduction of the commutativity problem to SMT solving, which is presented in Section 4.2, concentrates on the influence of the input on the loop s_2 only, we need to transform the reducer such that any input happens in the loop only.

To illustrate the issue, we consider the reducer shown in Figure 3. The reducer presented in the figure remembers the first input value in the variable m, increases its value by 10, and then updates its value to bigger ones if any occur in the loop. The main loop of the reducer is commutative in this case, but the reducer is not commutative. A counterexample can be easily found. With the list $[1,2,3,4,5]$ and its permutation $[5,4,3,2,1]$ as the inputs, the reducer outputs 11 and 15, respectively.

```
m := cur() + 10;
Loop{
|   t := cur()
|   if t > m then
|   |   m := t
}
; out(m)
```

Fig. 3: The $\max^+$ reducer with input before the main loop.

Our transformation will handle the example from Figure 3 as follows. We move the prefix s_1 into the loop body and use a new variable s to force that the execution of s_1 is always before the original loop body. The result after the transformation is demonstrated in Figure 4. The new reducer program has the same inputs/outputs as the original one. Therefore, if the new reducer is commutative, the original one is also commutative.

In general, the problem with the input before the loop can be handled as follows, including the case where $cur()$ occurs multiple times in s_1. Assume that the s_1 part of the reducer $s_1;\textbf{Loop}\{s_2\};s_3$ has the form $c_0;x_1{:=}cur();c_1;\ldots;x_m{:=}cur();c_m$ where $cur()$ does not occur in $c_0,c_1,\ldots,c_m$. In the transformed reducer, the part before the loop will be $c_0;s := 1$, and the loop body will contain several new branch statements. In particular, for all $j \in [1,m]$, we add the branch statement **if** $s = j$ **then** $x_j := cur();c_j;s := s+1$. Moreover, we transform the original loop body into the branch **if** $s = m+1$ **then** s_2.

```
s := 1;
Loop{
    if s = 1 then
        m := cur() + 10; s := 2
    else
        t := cur()
        if t > m then
            m := t
}
; out(m)
```

Fig. 4: The $\max^{+\texttt{fix}}$ reducer.

4.2 The Prover: Reduce Commutativity Checking to SMT Solving

After the transformation described above, the reducer $s_1;\textbf{Loop}\{s_2\};s_3$ never calls the $cur()$ function in the s_1 part before entering the loop. Further, we assume w.l.o.g. that the reducer reads exactly one input in one loop iteration. When multiple reads from the input occur in a single execution path from the begin to the end of the loop body, we can use additional variables and branch statements to break the path into several auxiliary ones, each reading just once.

The command s_2 can be viewed as a function F that reads the values of all variables and the current input before executing s_2 and outputs the values of all variables after s_2. Note that s_2 contains no nested loop structure. Hence, a bounded summary in quantifier-free linear integer arithmetic is sufficient for describing F.

Formally, the function $F(n,x_1,x_2,\ldots,x_k) : \mathbb{Z}^{k+1} \to \mathbb{Z}^k$ returns a tuple of values $x_1',x_2',\ldots,x_k'$ where n is the current input value of the reducer, x_i and x_i' are the values of the variables before and after the execution of s_2, respectively, for $i \in [1,k]$. The construction of F from s_2 can be done in the standard way.

We reduce the reducer commutativity verification to checking validity of the following formula for all possible values of $n_1,n_2,x_1,x_2,\ldots,x_k$:

$$F(n_1,F(n_2,x_1,x_2,\ldots,x_k)) = F(n_2,F(n_1,x_1,x_2,\ldots,x_k)). \tag{1}$$

Intuitively, the formula says that starting from the same initial valuations of variables and with two different input orders, $[n_1;n_2]$ and $[n_2;n_1]$, the values of all program variables are the same after we execute the loop body twice. The first execution reads n_1 and then reads n_2. The other execution reads the two inputs in the reverse order. Since any permutation of the input can be obtained by a sequence of permutations of neighbouring inputs, the validity of Formula 1 implies that the permutations will not change the final variable valuation and hence the output in s_3.

Consider the reducer computing the average value (Figure 1) as an example. The reducer has two variables s and c. We get that $F_{average}(n,s,c) = (s+n,c+1)$. In this case, Formula 1 is valid since $F_{average}(n_1,F_{average}(n_2,s,c)) = F_{average}(n_1,s+n_2,c+1) =$

$(s+n_1+n_2, c+2) = F_{average}(n_2, s+n_1, c+1) = F_{average}(n_2, F_{average}(n_1, s, c))$. This implies that the reducer is commutative.

A note on dealing with output in the main loop. So far we have assumed that there was no output in the main loop and mentioned that this restriction is lifted in J-ReCoVer. Due to space restrictions, a proper explanation of the way of handling this issue is beyond the scope of this paper, but we give at least a brief sketch of the solution. In particular, the Preprocessor performs one more transformation which adds an assignment $v := e$ for every $out(e)$ statement in the main loop where v is a fresh variable assigned just once in the loop body. This makes the output visible for our analysis since v appears in Formula 1. The Prover then makes an additional check whether the value of $F(n, x_1, x_2, \ldots, x_k)$ projected on v stays the same for any input value n and any initial values of the variables x_i.

4.3 An Optimisation by Live Variable Analysis

We now explain how a simple live variable analysis is used in J-ReCoVer to significantly improve the precision of commutativity checking.

In our initial experiments, we realised that Formula 1 is too strong, too often violated by reducers that are commutative. To illustrate the issue, we present a simple example. In the loop body, the input is first stored in a variable t and this is then assigned to s, i.e., $t := cur(); s := s+t$. After the loop, the value of s is output. In this case, the function F returns the updated values of both s and t after the execution of the loop body. Observe that $F(c_1, F(c_2, s, t)) = (s+c_1+c_2, c_1)$ and $F(c_2, F(c_1, s, t)) = (s+c_1+c_2, c_2)$. It follows that Formula 1 is invalid. The second component of the returned tuples, which causes the invalidity, corresponds to the value of t after executing the loop body twice. Their values are c_1 and c_2, respectively, in $F(c_1, F(c_2, s, t))$ and $F(c_2, F(c_1, s, t))$. However, in this case, the value of t will not affect the output of the reducer.

To handle the above issue, we perform a simple backward live variable analysis to collect all variables whose value may propagate to the output command after the loop execution. Only these variables are then required to be equivalent. For the example above, the variable t will be ignored in the equivalence checking and hence the program can be proven commutative. In our evaluation, the ratio of reducers that our approach can successfully analyse is significantly increased—in particular, from 6.8 % to 97.5 %—by using this optimisation.

5 Evaluation

J-ReCoVer is implemented in Java and built on top of Soot 2.5.0 and Z3 4.7.1. We ran J-ReCoVer on a virtual machine with 4GB of memory running Ubuntu 16.04.5 LTS on a server with AMD Opteron 6376 CPU.

Benchmark collection In order to properly evaluate the performance of J-ReCoVer, we used the search engine *searchcode.com* to collect Java programs containing the key strings "public void reduce" or "protected void reduce". Since there is an upper bound on the number of results returned from the search engine, we added different search filters in order to get more data. We tried all 12 combinations of six filters on the code length $\{< 50,\ 50 \sim 250,\ 250 \sim 450,\ 450 \sim 650,\ 650 \sim 850,\ 850 \sim 1050,\ 1050 \sim 1250\}$ and two filters for data sources $\{github.com, bitbuckect.com\}$. In total, we got $11,346$ Java programs. We excluded cases that were not Hadoop MapReduce reducer programs (those do not import the Hadoop library, do not extend or implement the reducer interface) and obtained $1,273$ examples. We further removed duplicates, those that could not be compiled, and those with non-numerical data types (e.g., strings). We obtained 118 reducers as the final benchmarks. Table 1 contains more details of the considered reducer functions.

Table 1: Size of the reducers.

	Line	Variable	Branch
Min.	5	4	0
Avg.	20.5	14.7	1.2
Max.	58	37	5

Results J-ReCoVer successfully handled 115 cases (97.5 %) out of the considered ones. Among them, 106 cases are commutative, while 9 are not. The analysis time ranged between 9.8 and 8.6 seconds. On average, 72 % of the execution time was spent in compiling Java source code to bytecode, which is the input of the Soot tool. Further, 27 % of the execution time was spent in the Preprocessor, in which Soot is used to transform Java bytecode to Jimple and perform the program transformation. The time spent in Solver is quite limited (< 1 %) since the real-world integer reducer programs are usually not that big.

There is no other tool that could handle the reducers as they are. Perhaps some other tools could be applied on the transformed programs, but the transformation would still be needed, and our SMT-based back-end verifier (the Solver) turned out to work efficiently. Hence, we did not feel a need to replace it by another verification tool. Of course, in the future, this can be done if need be.

J-ReCoVer failed in three cases out of the considered ones because the three reducers use more complicated control structures than what J-ReCoVer currently supports. Namely, they use a branch statement before entering the loop, i.e., they have the form of **if** g **then** $(s_1; \mathbf{Loop}\{s_2\}; s_3)$ **else** $(s'_1; \mathbf{Loop}\{s'_2\}; s'_3)$. In theory, such a program can be handled by more sophisticated program transformation. For example, we can merge the two loops and push the outer branch condition into the merged loop. Extensions of J-ReCoVer to be able to handle such constructions, together with a support for more data types, is among our future directions.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y.F., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: TACAS. (2012) 204–219
2. Abdulla, P.A., Atig, M.F., Chen, Y.F., Leonardsson, C., Rezine, A.: Memorax, a precise and sound tool for automatic fence insertion under TSO. In: TACAS. (2013) 530–536
3. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: FM. (2011) 200–214

4. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: ESOP. (2013) 533–553
5. Bouajjani, A., Emmi, M., Enea, C., Ozkan, B.K., Tasiran, S.: Verifying robustness of event-driven asynchronous programs against concurrency. In: ESOP. (2017) 170–200
6. Chen, Y.F., Hong, C.D., Sinha, N., Wang, B.Y.: Commutativity of reducers. In: TACAS. (2015) 131–146
7. Chen, Y.F., Lengál, O., Tan, T., Wu, Z.: Register automata with linear arithmetic. In: LICS. (2017) 1–12
8. Chen, Y.F., Song, L., Wu, Z.: The commutativity problem of the MapReduce framework: A transducer-based approach. In: CAV. (2016) 91–111
9. Csallner, C., Fegaras, L., Li, C.: New ideas track: testing mapreduce-style programs. In: FSE. (2011) 504–507
10. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI. (2004)
11. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Automated discovery of simulation between programs. In: LPAR. (2015) 606–621
12. Inverso, O., Tomasco, E., Fischer, B., Torre, S.L., Parlato, G.: Bounded model checking of multi-threaded c programs via lazy sequentialization. In: CAV. (2014)
13. Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification of pointer programs by predicate abstraction. Formal Methods in System Design **52**(3) (2018) 229–259
14. Lal, A., Reps, T.: Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In: CAV. (2008)
15. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: TACAS. (1998) 151–166
16. Smith, C., Albarghouthi, A.: Mapreduce program synthesis. In: PLDI. (2016) 326–340
17. Xiao, T., Zhang, J., Zhou, H., Guo, Z., McDirmid, S., Lin, W., Chen, W., Zhou, L.: Nondeterminism in mapreduce considered harmful? In: ICSE. (2014) 44–53