IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 16, 2021, accepted October 8, 2021, date of publication October 15, 2021, date of current version November 9, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3120414

Consistency Validation Method for Java Fine-Grained Lock Refactoring

YANG ZHANG, CHUNXIA LI, AND YU BAI
School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang, Hebei 050018, China

Corresponding author: Yu Bai (baiyu@hebust.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61902108, in part by the Scientific Research Foundation of Hebei Educational Department under Grant ZD2019093, and in part by the Natural Science Foundation of Hebei Province under Grant F2019208305.

ABSTRACT Many existing refactoring tools reduce the possibility of lock conflicts and improve the concurrency of the system by reducing lock granularity and narrowing the scope of locked objects. However, such refactorings can lead to changes in concurrent program behavior, introduce concurrency errors, and often even produce code that does not compile or can be compiled but has changed semantics. To address the problem of changes in concurrent program behavior caused by transferring from coarse-grained locking to fine-grained lock refactoring, a refactoring consistency validation method for fine-grained locking is proposed. Firstly, the types of behavioral changes caused by the existing refactoring engine are analyzed in terms of thread interactions. Secondly, the relevant consistency checking rules are summarized according to the types. Finally, with the help of various program analysis techniques such as call graph analysis, alias analysis and side-effect analysis, the corresponding checking algorithms are designed according to the consistency checking rules to check the consistency of the program before and after refactoring. We implement an automatic validation tool as an Eclipse plug-in. Our approach is verified by ten open-source projects including HSQLDB, Xalan and Cassandra, etc. A total of 1,483 refactoring methods were tested, and 60 inconsistent synchronization behaviors were found, which improved the robustness of refactoring in terms of data dependence and execution order.

INDEX TERMS Fine-grained lock, refactoring, consistency validation, alias analysis, side-effect analysis.

I. INTRODUCTION

Locks are used to control access to shared resources by multiple threads and are one of the most commonly used synchronization methods, but the use of locks can easily lead to lock contention. Lock contention is a phenomenon in which multiple threads attempt to access a shared resource protected by the same mutually exclusive lock during program execution. In a highly concurrent environment, especially when the critical section is large or when threads enter it frequently, the performance degradation caused by lock contention can be significant. A critical section is a program fragment that accesses a shared resource that cannot be accessed by multiple threads at the same time. One important property of a lock is the lock granularity. Lock granularity is a measure of the amount of data protected by a lock. A reasonable locking granularity maximizes the use of shared resources, so it is important to optimize the locking granularity. Coarse-grained locking is a type of lock with a small number of locks and a large amount of data protected by each lock. Fine-grained locks are locks with a large number of locks, each protecting a small amount of data. Usually, when choosing coarse-grained locks, the lock overhead is low when accessing protected data in a single thread, but the performance is poor when multiple threads access it at the same time because of the increased lock contention. Conversely, using fine-grained locks increases lock overhead and reduces lock contention.

Due to the difficulty of developing concurrent programs, program developers tend to use coarse-grained locking methods to reduce the burden, such as synchronous methods or synchronous blocks in Java. However, coarse-grained locking may actually result in many operations being executed sequentially, reducing the efficiency of the program. In order to optimize synchronization, developers have introduced fine-grained synchronization mechanisms to reduce the locking granularity and narrow the range of locked

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

objects, reducing the possibility of lock conflicts. Since the read operation itself does not affect the data integrity and consistency, in suitable scenarios, using the read-write split lock to replace the exclusive lock and partition the system function points can effectively improve the concurrency of the system. Among the fine-grained lock refactoring tools, FineLock [1] adopts the read-write split lock approach and implements a kind of automatic fine-grained lock refactoring based on a push-down automaton to complete the automatic conversion from coarse-grained locks to fine-grained locks, which improves the refactoring efficiency compared with manual refactoring.

In existing automated refactoring, improperly refining coarse-grained locks can aggravate the uncertainty and concurrency during the running of concurrent programs, and the refactored programs may have new concurrent behaviors, such as deadlock, live lock, data contention, etc. For sequential programs, developers are usually encouraged to use regression testing techniques to ensure that refactoring does not change the program behavior [2]. However, this approach highlights its effectiveness when only very few threads are scheduled in a concurrent environment.

There are many studies on the correctness of refactoring concurrent programs. Some researchers believe that preconditions can be set at the beginning of refactoring to prevent inconsistent behavior, but automated refactoring tools cannot check all preconditions for refactoring [3], [4], so they cannot completely rely on checking preconditions to complete all consistency verification. In addition, modular reasoning [5] and framework protection behavior [6] have been proposed to improve the correctness of concurrent programs. Among the consistency detection tools for software refactoring, the representative ones are Randoop [7]–[9] and EvoSuite [10], which are not only able to detect errors introduced by refactoring, but also can help improve the efficiency of refactoring. The refactoring detection method IDiff [11] can effectively detect differences, moves and refactoring-related changes in the source code and help to understand software evolution. The model refactoring checking tool CVT [12] helps to solve the problem of checking the consistency between the model and its evolution. There have been some works on concurrent program refactoring to detect inconsistent behavior before and after refactoring [13], [14], but the correctness verification in fine-grained lock refactoring has yet to be studied in depth. For example, although FineLock has a certain consistency detection mechanism, it still lacks treatment for ensuring data synchronization and consistency of execution order. Using consistency checking methods to examine the synchronization behavior before and after fine-grained lock refactoring and the original external characteristics of the program allows software developers and maintainers to gain more insight into the evolutionary history of the software and thus better maintain it.

To check the synchronization change of behavior caused by fine-grained lock refactoring, the paper proposes a method to check the consistency of refactoring. We analyzed three kinds of behavior changes caused by the existing refactoring engine and summarized the relevant consistency validation rules. According to the proposed rule, variable overlapping validation, condition missing validation and sequential violation validation are designed to verify the consistency before and after refactoring. In experiments, we perform fine-grained lock refactoring of the benchmark program by FineLock refactoring tool, and then use the program before and after refactoring as input to perform consistency checking. We validated the tool on ten large scale real applications and detected a total of 1483 refactorings for read-write lock separation and found 60 inconsistent synchronization behaviors. Experimental results show that the method proposed can effectively discover the inconsistent synchronization behavior caused by fine-grained lock refactoring. The main contributions of the paper include the following:

- We found that the existing refactoring technology may introduce concurrency errors that could change the order of threads (see §3).
- We proposed a refactoring consistency validation method oriented to fine-grained locks (see§4).
- We developed an automated verification tool implemented as Eclipse plugins(see§5).
- We evaluated our tool on several real-world applications(see§6).

Finally, we conclude.

II. RELATED WORKS

The paper implements a refactoring consistency validation tool for fine-grained locks. We mainly focus on two aspects of related work: fine-grained lock refactoring and refactoring verification.

A. FINE-GRAINED LOCK REFACTORING

In the area of concurrent code optimization through refactoring, Schäfer, in collaboration with IBM T. J. Watson Research Center, designed a refactoring tool for Java display locks, Relocker [15], [16], to refactor synchronous locks into Reentrant locks and refactor Reentrant locks into Read-write locks. Tao and Qian [17] proposed an automatic lock decomposition refactoring method for Java programs to divide lock protection domains based on class attribute domains, and implemented the automatic refactoring tool in the form of an Eclipse plug-in. Yu and Pradel [18] proposed a lock decomposition method in their research on optimizing synchronization bottlenecks, which reconstructs lock dependencies and uses fine-grained locks to protect disjoint sets of shared variables. Zhang *et al.* [19] proposed FineLock, an automatic refactoring method for fine-grained locks, which uses lock degradation and lock decomposition to achieve a fine-grained way of protecting critical sections.

The above studies implement the fine-grained protection of the critical section by lock allocation, lock reservation, atomic block and lock degradation, lock decomposition and other techniques to reduce the critical section competition,

our research is to check the consistency of the change of synchronization behavior before and after the fine-grained lock refactoring, mainly for lock decomposition and lock degradation.

B. REFACTORING VERIFICATION

In terms of consistency verification of refactorings, Ubayashi *et al.* [20] proposed the concept of RbC (Refactoring by Contract), a contract-based technique for verifying refactorings, in order to deal with the bugs embedded in refactorings of cut-oriented programming. The contract in RbC consists of preconditions, postconditions and invariants. After the introduction of RbC, it is checked whether the refactoring preserves the behavior and whether it actually improves the internal structure. Yin *et al.* [21] proposed a new method for formal verification of the functional correctness of software, Echo, which can be used to verify refactorings. Echo mainly proves that the semantics of the refactored program is equivalent to that of the original program. Garrido and Meseguer [22] specified three useful Java refactorings, giving detailed correctness proofs for two of them. Each of these methods defines some specifications and conditions to verify the correctness of the refactorings.

Software refactoring is the modification of software to improve its structure, clarity, extensibility and reusability without changing its functionality and external visibility. Therefore, it is necessary to check the consistency of the program before and after refactoring to verify the functionality and external visibility of the program after refactoring. In terms of consistency testing for sequential programs, Silva *et al.* [10] proposed a regression test suite to verify refactorings, and regression tests can also be used for consistency checking before and after refactorings. Abadi *et al.* [23] proposed a method for verifying parallel code after refactoring, which is based on symbolic interpretation, which utilizes the original sequential code that has been tested and verified in most cases and checks whether it is equivalent to that code after refactoring. Dao *et al.* [11] proposed a tool for consistency checking of behavior in model refactoring. In the area of consistency checking for concurrent programs, Hofer *et al.* [24] proposed a new approach to analyze lock contention in Java applications by tracking locking events in the Java virtual machine, which reveals the causes of lock contention and identifies performance bottlenecks of locks. Schäefer *et al.* [25] proposed a behavior-preserving technique to avoid changing program behavior. By analyzing the possible causes of inconsistent behavior changes due to current refactorings, a concurrent program behavior preservation technique is proposed to address the causes of the problem. The technique introduces synchronization dependencies and simulates the ordering constraints of the Java memory model, and proves that the technique can guarantee the behavior retention. Silva *et al.* [12] proposed a refactoring consistency detection method for concurrent software refactoring, which uses control flow analysis and data flow analysis to detect changes before and after refactoring, and synchronization dependency analysis to detect changes in synchronization dependencies before and after refactoring.

The above studies implement refactoring correctness checking in various forms and automate the tools well, but they are all performed for concurrent programs, and we focus on consistency checking for refactored programs with fine-grained locks.

III. MOTIVATION

This section shows code snippets of existing FineLock refactoring tools that change the behavior of programs before and after software refactoring, giving the motivation for the paper.

To illustrate the change in synchronous behavior of the program before and after the refactoring, the code structure is illustrated. Figure 1 is a selection from the Guava API documentation and shows the refactoring that splits a critical area in Figure 1(a) into a critical area in Figure 1(b) that is locked by a write lock and a read lock, respectively. If there are two threads executing this code at the same time, the synchronized modification in Figure 1(a) ensures that only one thread is accessing the get() method at the same time, the two threads return the result value of value, null and multiple runs remain consistent. After refactoring, if the current thread2 is acquiring the write lock when the write lock has been acquired by thread1, the current thread enters the wait state. When thread1 releases the write lock, the Java memory model will refresh the shared variable value of the local memory corresponding to the thread to null in the main memory. If thread2 gets the write lock before thread1 gets the read lock, the original operation semantics may be changed if both return null.

Figure 2 shows two implementations of the processCached() method, which is a typical cache processing operation taken from the Java API documentation for read/write locks. The method processCached() simulates the operation on the database and the cache by first determining whether the data exists in the cache, and if so, reading the data directly from the cache, otherwise writing the data from the database to the cache.

In Figure 2(a), the method uses synchronized for synchronization control, and the whole method is under the protection of the lock is a coarse-grained protection. Figure 2(b) is a fine-grained locking method, which first obtains the read lock and judges the cacheValid (lines 3-4), if the if condition does not hold, it directly reads and releases the read lock (lines 15-17). If it holds, it releases the read lock to obtain the write lock (lines 5-6), and when the cache is written from the database, it obtains the read lock and then releases the write lock to complete the lock degradation operation (lines 8-11). However, after refactoring, the conditional statement and its statement body are refactored to different critical areas. If there are two threads executing this code at the same time, even if the data is not in the cache at the beginning, both threads may read the current state of the cache and store it in their respective CPU cache, and if the state is not rechecked,

```
1. public class SafeBox<V>{
2.     private V value,result;
3.     synchronized V get() {
4.         result=value;
5.         value=null;
6.         return result;
7.     }
8. }
```

(a)Before refactoring

```
1. public class SafeBox<V>{
2.     ReentrantReadWriteLock    rwlock=new ReentrantReadWriteLock();
3.     private V value,result;
4.     V get(){
5.         rwlock.writeLock().lock();
6.         try {
7.             result=value;
8.             value=null;
9.         }finally {
10.            rwlock.writeLock().unlock();
11.        }
12.        rwlock.readLock().lock();
13.        try {
14.            return result;
15.        }finally {
16.            rwlock.readLock().unlock();
17.        }
18.    }
19. }
```

(b)After refactoring

FIGURE 1. Example of fine-grained lock decomposition for program behavior changes.

```
1. public synchronized void processCached() {
2.         if (!cacheValid) {
3.             … // write to cache
4.         }
5.         … // read data
6.   }
```

(a)Before refactoring

```
1.  ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
2.  public void processCached() {
3.          rwl.readLock().lock();
4.          if (!cacheValid) {
5.              rwl.readLock().unlock();
6.              rwl.writeLock().lock();
7.              try {
8.              … // write to cache
9.                  rwl.readLock().lock();
10.             } finally {
11.                rwl.writeLock().unlock();
12.             }
13.         }
14.         try {
15.             … // read data
16.         } finally {
17.             rwl.readLock().unlock();
18.         }
19. }
```

(b)After refactoring

FIGURE 2. An example of fine-grained lock downgrading to change program behavior.

another thread may obtain the write lock and change the state before.

Because of the exclusivity of the synchronized keyword, all threads must pass through the shared area protected by synchronized serially. Therefore, in Figure 3(a), method m() is executed with flag1 and flag2 read directly after self-incrementing, and if the if condition holds, the bug string is output. When method m1() is executed, flag1 is also read directly after the assignment operation, and if the if condition is valid, the bug string is output. Figure 3(b) shows the refactored code. In method m(), firstly, a write lock is applied to the self-increment operation (lines 7-12), and then a read lock is applied to the if condition statement and the output statement (lines 14-20). In method m1(), the assignment operation is first locked with a write lock (lines 24-28), and then the if condition statement and the output statement are locked with a read lock (lines 30-36). After refactoring, the lock object of method m() is rwlock and the lock object of method m1() is nulock. If a thread executes the method in class C, the execution order CS1->CS3->CS2->CS4 may occur, making the program unable to output the bug string.

From the above example, we can see that the refactoring uses lock downgrading and lock decomposition to achieve fine-grained protection of the critical section, which reduces lock contention to some extent, but may lead to changes in synchronization behavior due to improper lock downgrading and lock decomposition, which may lead to program errors.

IV. CONSISTENCY VALIDATION RULES

The paper focuses on the consistency checking operation of FineLock, a fine-grained lock refactoring tool that uses a push-down automaton to construct different lock patterns. Although consistency detection rules are defined in FineLock to constrain the refactoring of lock degradation

```java
1. public class C {
2.   int flag1=0;
3.   int flag2=0;
4.   public synchronized void m() {
5.     flag1++;
6.     flag2++;
7.     if(flag1==flag2) {
8.       System.out.println("bug");
9.     }
10.  }
11.  public synchronized void m1() {
12.    flag1=0;
13.    if(flag1==0) {
14.    System.out.println("bug");
15.    }
16.  }
17.}
```

(a)Before refactoring

```java
1. public class C {
2.   int flag1=0;
3.   int flag2=0;
4.   ReentrantReadWriteLock rwlock= new
         ReentrantReadWriteLock();
5.   ReentrantReadWriteLock nulock= new
         ReentrantReadWriteLock();
6.   public void m() {
7.     rwlock.writeLock().lock();
8.     try {
9.         flag1++;
10.        flag2++;
11.    }finally {
12.        rwlock.writeLock().unlock();
13.    }
14.        rwlock.readLock().lock();
15.        try {
16.            if(flag1==flag2) {
17.            System.out.println("bug");
18.            }
19.        }finally {
20.          rwlock.readLock().unlock();
21.        }
22.  }
23.  public void m1(){
24.     nulock.writeLock().lock();
25.     try{
26.         flag1=0;
27.      }finally{
28.         nulock.writeLock().unlock();
29.      }
30.         nulock.readLock().lock();
31.         try{
32.             if(flag1==0) {
33.               System.out.println("bug");
34.             }
35.         }finally{
36.             nulock.readLock().lock();
37.         }
38.   }
39. }
```

(b)After refactoring

FIGURE 3. Example of fine-grained lock decomposition for program behavior changes.

and lock decomposition, there are still inconsistent behaviors as described in the previous section. In order to further ensure the correctness of refactoring, we provide an additional description of FineLock's consistency checking rules.

Definition 1 (Invariance of the External Behavior of the Program Before and After Refactoring): The application P before and after refactoring is denoted as P_{before} and P_{after} respectively, The external behavior of the program is expressed as $Behavior\ (P)$, and the external behavior unchanged before and after refactoring is defined as $Behavior\left(P_{before}\right) \equiv Behavior\left(P_{after}\right)$, otherwise it is defined as $Behavior\left(P_{before}\right) \neq Behavior\left(P_{after}\right)$.

Definition 2 (Set of Critical Sections): The set of all critical sections in an application P is defined as $C = \{c_1, c_2, \ldots, c_n\}$, and the number of elements in the set C is denoted by $|C|$, $n = |C|$. The set of critical sections of P_{before} and P_{after} is correspondingly expressed as C_{before} and C_{after}. Since the critical section needs to be divided during the refactoring process of FineLock from coarse-grained lock to fine-grained lock, $\left|C_{before}\right| < \left|C_{after}\right|$. Since the code in an application P is finite, it can be known that the critical section contained in P is also finite, so C is a finite set.

Definition 3 (Critical Section Division) For $\forall c_i \in C\ (1 \leq i \leq n)$, a coarse-grained critical section c_i can be divided into k sub-critical sections, defined as $c_i = \{c_{i1}, c_{i2}, \ldots c_{ik}\}$.

Definition 4 (Set of Read and Write Operations in Critical Section) The set of read and write operations of all critical section C in a program P is defined as $OP = \left\{OP_1, OP_2, \ldots, OP_y\right\}$. For $\forall c_i \in C\ (1 \leq i \leq n)$, c_i can contain several sets of read and write operations OP_i, $OP_i = \{op_{i1}, op_{i2}, \ldots op_{ir}\} \subseteq c_i$. Correspondingly, the read operation is denoted as op^r_{i1}, and the write operation is denoted as op^w_{i1}.

Definition 5 (Set of Locks Before Refactoring): The set of all locks used to protect the critical section C_{before} in an application P_{before} is defined as the set $S = \{s_1, s_2, \ldots s_m\}$.

Due to C is a finite set, the code corresponding to the critical section is usually protected by a lock. In theory, a critical section can be protected by multiple locks, but a critical section can be protected by no more than two locks in practice, so S is a finite set. For $\forall s_e \in S\ (1 \leq e \leq m)$, s_e means at it contains both locking and unlocking
operations.

Definition 6 (Lock Protection): For $\forall c_k \in C\ (1 \leq k \leq n)$, $\exists s_e \in S\ (1 \leq e \leq m)$, if the critical section c_k is in the protection of the lock s_e, the lock protection relationship is defined as $s_e \oplus c_k$.

A lock can protect multiple critical sections. The set of these critical sections is c_k. c_k is a subset of C_{before}, that is, $c_k \subseteq C_{before}$, then the lock protection relationship is defined as $s_e \oplus c_k$.

Definition 7 (Refactored Lock Set): In the refactored application P_{after}, the set of all lock used to protect the critical section C_{after} is defined as set $L = \{l_1, l_2, \ldots l_t\}$.

For $\forall l_a \in L\,(1 \leq a \leq t)$, c_v is a subset of C_{after}, that is, $c_v \subseteq C_{after}$, the lock protection relationship is defined as $l_a \oplus c_v$. Definition 7 illustrates the difference between the lock set before and after refactoring. After FineLock refactors the synchronization lock into a read-write lock, the lock set is represented by L.

Definition 8 (Conditional Layout): For $\forall c_i \in C\,(1 \leq i \leq n)$, $\exists z_j \in Z$, $z_{j-e} \in Z_e$, where Z is a conditional control set, Z_e is a conditional end instruction set, z_j, z_{j-e} are a certain control condition and condition end instruction respectively. If z_j, z_{j-e} lie in the critical section c_i at the same time, it is defined as $z_j \cup z_{j-e} \rhd c_i$. Conversely, it is defined as $z_j \cup z_{j-e} \not\rhd c_i$.

Definition 9 (Happens-Before Relation): For $\forall c_i \in C$, $\{op_{i1}, op_{i2}, \ldots op_{ir}\} \subseteq c_i\,(1 \leq i \leq n)$, $\exists u, v\,(1 \leq u \leq r, 1 \leq v \leq r)$ the Happens-before relation is defined as $op_{iu} \rightarrow op_{iv}$ if op_{iu} occurs before op_{iv}. By virtue of the transferability of the Happens-before relation, if $op_{iu} \rightarrow op_{iv}$ and $op_{iv} \rightarrow op_{iw}$, then $op_{iu} \rightarrow op_{iw}$.

The Happens-before relationship is defined as a sequential relationship between read and write operations based on the Java memory model and is an important criterion for memory consistency in the Java language. One way to establish this relationship is through a synchronization relationship in the program, where the operation before unlocking occurs before the operation after unlocking to obtain the lock.

Based on the above definition, the consistency validation rules for fine-grained lock refactoring are given below.

Rule 1 (Correspondence of Locks Before and After Refactoring): If we want to guarantee $Behavior\,(P_{before}) \equiv Behavior\,(P_{after})$, before refactoring for $\forall c_k \in C\,(1 \leq k \leq n)$, $\exists s_e \in S\,(1 \leq e \leq m)$, there is $s_e \oplus c_k$. After refactoring, for $\forall c_v \in C\,(1 \leq v \leq n)$, $\exists l_a \in L\,(1 \leq a \leq t)$, $l_a \oplus c_v$. If $c_v \subseteq c_k$, there is a one-to-one correspondence between s_e and l_a, denoted as $s_e \leftrightarrow l_a$.

Our consistency validation method focuses on fine-grained locks refactoring, after the refactoring the critical sections will still in locks protection. Therefore, it will change the program behavior if the critical sections are without locks protection after the refactoring. Rule 1 illustrates that the critical area, which is in lock protection before refactoring, remains in lock protection after refactoring, and there is a one-to-one correspondence between the lock before refactoring and the lock after refactoring. If rule 1 is broken, the consistency will be violated, but the violation of consistency is not necessarily caused by breaking rule 1. Rule 1 is a necessary and insufficient condition for behavior consistent of the program before and after refactoring, i.e. Rule1 $\Leftarrow Behavior\,(P_{before}) \equiv Behavior\,(P_{after})$, but Rule1 $\not\Rightarrow Behavior\,(P_{before}) \equiv Behavior\,(P_{after})$; $\neg$ Rule1 $\Rightarrow Behavior\,(P_{before}) \neq Behavior\,(P_{after})$.

Rule 2: Before refactoring, for $\forall c_i \in C$, $\{op_{i1}, op_{i2}, \ldots op_{ir}\} \subseteq c_i\,(1 \leq i \leq n)$, $\exists s_e \in S\,(1 \leq e \leq m)$, there is $s_e \oplus c_i$. After refactoring, $c_i = \{c_{i1}, c_{i2}, \ldots c_{ik}\}$, if $\exists l_a \in L\,(1 \leq a \leq t)$, there is $s_e \leftrightarrow l_a$, and for $\forall c_{ix}, c_{iy}$ $(1 \leq i \leq n, 1 \leq x \leq k, 1 \leq y \leq k, x \neq y)$, there is $l_a \oplus \{c_{ix}, c_{iy}\}$. For $\forall op_{ip}, op_{iq}\,(1 \leq p \leq r, 1 \leq q \leq r)$, if $\{op_{ip}\} \subseteq c_{ix}$, $\{op_{iq}\} \subseteq c_{iy}$, op_{ip} and op_{iq} access the same memory location, and one of them is a write operation $op_{ip}^{w} \Big| op_{iq}^{w}$, after the lock is broken down, $Behavior\,(P_{before}) \neq Behavior\,(P_{after})$.

Rule 2 constrains the lock decomposition in terms of maintaining the atomicity of the original critical area to ensure that the atomicity of the original critical area is not broken. If there are op_{ip} and op_{iq} accessing the same memory location, it is possible that the original operation semantics may be changed due to thread interaction. It will cause problems of visibility, atomicity, ordering. Rule 2 is a necessary and insufficient condition for behavior violation of the program before and after refactoring, i.e. Rule 2$\Rightarrow$ $Behavior\,(P_{before}) \neq Behavior\,(P_{after})$.

Rule 3: For $\forall c_i \in C\,(1 \leq i \leq n)$, $\exists z_j \in Z, z_{j-e} \in Z_e$, there is $z_j \cup z_{j-e} \rhd c_i$. After the critical section is decomposed, $c_i = \{c_{i1}, c_{i2}, \ldots c_{ik}\}$, for $\forall c_{ix} \in C\,(1 \leq x \leq k)$, there is $z_j \cup z_{j-e} \not\rhd c_{ix}$ and there is no judgment statement identical to z_j in the sentence block judged by z_j, then $Behavior\,(P_{before}) \neq Behavior\,(P_{after})$ after lock decomposition.

Rule 3 illustrates that the original control condition and the conditional end instruction are in the same critical section. and after the decomposition of the critical section, it is possible to change the original operation semantics due to thread interaction if they are not in the same critical section and no secondary determination is made within the statement block. Rule 3 will cause problems of visibility, violate the consistency of the program. Rule 3 is a necessary and insufficient condition for behavior violation of the program before and after refactoring, i.e. Rule 3$\Rightarrow$ $Behavior\,(P_{before}) \neq Behavior\,(P_{after})$.

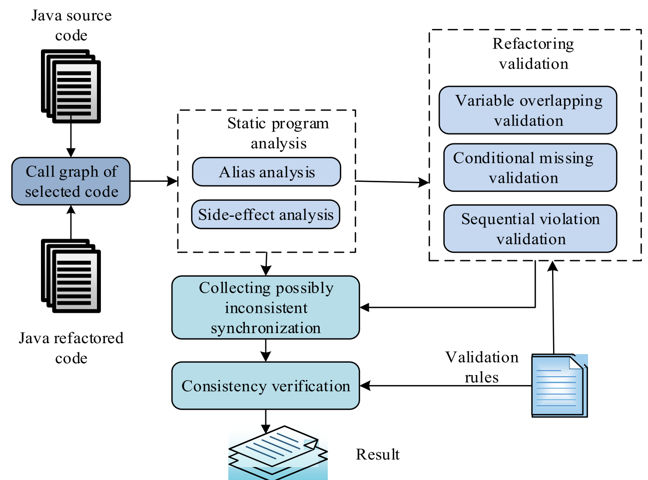

FIGURE 4. The validation framework.

Rule 4: Before refactoring, for $\forall c_i \in C\ (1 \le i \le n)$, $\exists s_e \in S\ (1 \le e \le m)$, there is $s_e \oplus c_i$. After refactoring $c_i = \{c_{i1}, c_{i2}, \ldots c_{ik}\}$, if $\exists l_a \in L\ (1 \le a \le t)$, such that $s_e \leftrightarrow l_a$, and for $\forall c_{ij}\ (1 \le i \le n, 1 \le j \le k)$, there is $l_a \oplus \{c_{ij}\}$. When $\{op_{i1}, op_{i2}, \ldots op_{ir}\} \subseteq c_i$, for $\forall op_{ip}, op_{iq}\ (1 \le p \le r, 1 \le q \le r)$, if $op_{ip} \rightarrow op_{iq}$ in C_{before}, then there is still $op_{ip} \rightarrow op_{iq}$ in C_{after}.

Rule 4 constrains the refactoring of lock decomposition by checking the Happens-before relationship of the read and write statements in the critical section to ensure that the relationship has not changed before and after refactoring. In the refactoring object of FineLock tool, the code before and after refactoring involves synchronization relationship, and the Happens-before relationship can be established on the basis of synchronization relationship, and then the rule can be determined. Happens-before is a necessary and insufficient condition for behavior consistent of the program before and after refactoring, i.e. $\neg$ Rule 4$\Rightarrow$ $Behavior\left(P_{before}\right) \neq Behavior\left(P_{after}\right)$.

V. CONSISTENCY VALIDATION METHOD

A. OVERVIEW

In the process of checking consistency, the source program C_{before} is firstly refactored with fine-grained locks to obtain the refactored program C_{after}. Based on the source code, we use the WALA [26] software analysis tool to generate the corresponding call graph and intermediate representation IR for the source program C_{before} and the refactored program C_{after} respectively. the analysis methods used are mainly alias analysis and side-effect analysis. Alias analysis is used to solve the alias problem of accessing variables. The side-effect analysis determines whether the relevant variables involved in the critical section have negative effects and generates read/write field sets. Finally, the variable overlapping validation, the conditional missing validation, and the sequential violation validation are designed to verify the consistency before and after refactoring according to the generalized consistency test rules. The variable overlapping validation is used to check whether the refactoring will destroy the data dependencies that existed before the refactoring, the conditional missing validation is used to analyze whether the refactoring will result in competing conditions, and sequential violation validation is used to analyze whether the refactoring will cause deviations in the execution order of threads. The validation framework is shown in Figure 4.

B. CALLGRAPH ANALYSIS

In the framework, the WALA analysis tool is used to generate a Call Graph for the source and refactored programs. In the specific generation process, we first obtain the object selected by the user in the inspection operation through Eclipse JDT, and then store the object in the analysis domain to build a class hierarchy, and finally generate a relational call graph based on the class hierarchy. In the implementation process, the program's relational call graph is obtained by implementing the makeCallGraph() method in the CallGraphBuilder interface in WALA. The call graph contains nodes and edges, where nodes represent methods and the edges represent the calling process between methods.

C. ALIAS ANALYSIS

Aliasing means that two access variables point to the same memory location and if one object value changes, the other will change accordingly.

Before performing the consistency check, all the methods in the program are first traversed through the call graph analysis, and the related variables involved in the synchronous method or synchronous block are collected.

When determining statement dependencies during the test, alias analysis of the relevant variables is required to determine whether the memory locations referred to by the two variable access operations are the same and to avoid misjudgment of dependencies. In the program, the aliasing statements may lead to two types of aliasing: assignment between object variables and combination of object type parameters during method calls. If two variables are aliased to each other and represented by a pair (var × var), the set of aliases is represented as [var × var]*. For example, if x, y are aliased to each other, they are denoted as (x × y) and are equivalent to (y × x).

D. SIDE-EFFECT ANALYSIS

The side-effects are defined as the modification of memory units during program execution. The side-effect analysis in the paper traverses and analyzes the intermediate representation IR in the method to determine whether the instruction modifies the memory units. The analysis algorithm is shown in Algorithm 1.

Algorithm 1 Side-Effect Analysis Algorithm

```
Input:  C – the target critical section
Output: Fprotected_read – read  field set
        Fprotected_write – write field set
1. Instructions ← all instructions from C;
2. Fprotected_write ← ∅, Fprotected_read ← ∅;
3. for each Ins in Instructions do
4.  sideEffectAnalysis(Ins, limit, Fprotected_write, Fprotected_read);
5. end for
6. void sideEffectAnalysis(Ins, limit, Fprotected_write, Fprotected_read)
7. if limit<5 then
8.   if  Ins is a write instruction to static field f ||
       Ins is a write instruction to instance field f then
9.     Fprotected_write ← Fprotected_write ∪ {f};
10.  else if Ins is a read instruction to field f then
11.    Fprotected_read ← Fprotected_read ∪ {f};
12.  else if  Ins is a InvokeInstruction then
13.    limit++;
14.    M ← method of Ins invoked;
15.    MInstructions ← all instructions from M;
16.  for each MIns in MInstructions do
17.  sideEffectAnalysis(MIns, limit, Fprotected_write, Fprotected_read);
18.  end for
19.  end if
20. end if
```

In the analysis of method call instructions, the number of method call entry levels is limited to 5 in order to ensure the execution efficiency of the tool. First, the instruction set

corresponding to the critical section is obtained and the sideEffectAnalysis method is called to analyze each instruction (lines 1-5). Second, the method layer limit is judged(line 7). After analyzing each instruction, if the instruction modifies the field, the field is written into Fprotected_write(lines 8-9). Finally, if it is a method call instruction, the counter of the call level is incremented by one(line 13)and the sideEffectAnalysis method is recursively called to analyze the instructions in the called method (lines 14-17). If the currently called method does not produce side-effects, the field is written to Fprotected_read (lines 10-11).

E. VARIABLE OVERLAPPING VALIDATION

According to Rule 2 mentioned in the previous section, we designed a variable overlapping validation, which mainly refers to the judgment of the relationship between statements based on data dependency. After fine-grained refactoring, if the dependent statements are distributed to the different critical sections, the execution order of the statements may be destroyed, resulting in changes in synchronization behavior. Algorithm 2 is the basic structure of variable overlapping validation.

Algorithm 2 Variable Overlapping Validation

```
Input: M/B – synchronization method/synchronization block
Output: DataDeMap – dependency set in M/B
1. C← critical section set in M/B
2. for each C_i in C do
3.   Instructions ← all instructions from C_i;
4.   Limit ← 0; WprotectedMap ← ∅; RprotectedMap ← ∅;
5.   for each Ins in Instructions do
6.     WprotectedMap(Ins) ← ∅, RprotectedMap(Ins) ← ∅;
7.     Fprotected_write ← ∅, Fprotected_read ← ∅;
8.     sideEffectAnalysis(Ins, limit, Fprotected_write, Fprotected_read);
9.     WprotectedMap(Ins)← WprotectedMap(Ins)∪ Fprotected_write;
10.    RprotectedMap(Ins)← RprotectedMap(Ins)∪ Fprotected_read;
11.    WprotectedMap ← WprotectedMap ∪ (Ins, WprotectedMap(Ins));
12.    RprotectedMap ← RprotectedMap ∪(Ins, RprotectedMap(Ins));
13. end for
14. for each key value Ins_i in WprotectedMap do
15.   for each key value Ins_j in RprotectedMap do
16.   if Ins_i ≠ Ins_j &&(WprotectedMap(Insi)∪ RprotectedMap(Insj))
          &&(RprotectedMap(Insi)∪ WprotectedMap(Insj))
          &&(WprotectedMap(Insi)∪ WprotectedMap(Insj))≠ Φ then
17.     DataDeMap ← DataDeMap ∪(Ins_i, Ins_j);
18.   end if
19.   end for
20. end for
21.   return DataDeMap;
22. end for
```

The algorithm 2 scans each synchronization method and synchronization block, and analyzes the fields used in them to determine the dependencies, and stores the read and write mapping relationships between statements and protected fields using two sets of key-value pairs. First, each critical section is analyzed and the number of calling layers, RprotectedMap and WprotectedMap are initialized (lines 4-5). Second, each instruction is traversed and the protected fields in the critical section are divided into protected read and protected write, which are respectively denoted as Fprotected_read and Fprotected_write. The read and write operation analysis of field f is performed on each instruction through the sideEffectAnalysis method (line 9). The protected read and protected write are mapped to the corresponding statements after side-effect analysis (lines 10-11). Finally, the statements with the same variables in the read-write mapping RprotectedMap and WprotectedMap are judged according to the dependency judgment rules (lines 15-17). since multiple critical sections may exist in a synchronous method or synchronous block after refactoring, the dependencies of multiple critical sections need to be merged (line 18).

F. CONDITIONAL-MISSING VALIDATION

Condition missing mainly checks the most common competing conditions, i.e. check first and execute later, by determining whether conditional statements and statement blocks are distributed to different critical sections after fine-grained lock refactoring, and the program does not redetermine the state. The conditional missing validation is shown in Algorithm 3.

First, the instruction set of the method is generated and two read lock instruction sets Rprotected, R1protected, write lock instruction set Wprotected, condition variable set IFprotected_read, read and write fieldset Fprotected_write, Fprotected_read, read lock counter and layer limit are initialized(lines1-4). If the statement is a read lock operation and the counter is zero, the statement protected by the read lock is written to Rprotected. If the instruction is a conditional judgment, the condition variable is written to IFprotected_read (lines 6-11). If the statement is a write lock operation and the counter is one, the statement protected by the write lock is written to Wprotected (lines 17-22) and each instruction is used to analyze the write operation of field f (line 21) in the sideEffectAnalysis method. If the instruction modifies the field, the field is written to Fprotected_write. If it is a method call instruction, the counter of the calling layer is incremented by one and the sideEffectAnalysis method is recursively called to analyze the write operation of the instruction in the called method. If the statement is a read lock operation and the counter is 1, the statement protected by the read lock is written to R1protected (lines 24-28). Finally, if there is a conditional judgment instruction in the first read lock critical section but no conditional end instruction, there is no conditional judgment instruction in the write lock critical section and there is no conditional judgment instruction in the second read lock critical section but there is a conditional end instruction, at the same time the condition variable is written (line 32), the method signature of the method is returned, otherwise, it returns null.

G. SEQUENTIAL VIOLATION VALIDATION

Sequential violation means that the refactoring may result in a rearrangement of the execution order of multiple critical sections due to changes in the locking objects of synchronous methods/synchronous blocks by fine-grained locking. when a thread releases a lock, the Java memory model flushes the shared variables in the local memory corresponding to the thread to the main memory according to the memory semantics of locks. If the critical sections (for example, if there

Algorithm 3 Conditional-Missing Validation

Input: M – synchronization method
Output: Sign/null – method signature
1. Instructions ← all instructions from M;
2. Rprotected ← ∅, Wprotected ← ∅, R1protected ← ∅;
3. IFprotected_read ← ∅;
 Fprotected_write ← ∅, Fprotected_read ← ∅;
4. count ← 0, limit ← 0;
5. **for** each Ins in Instructions **do**
6. **if**(Ins is a readlock instruction) &&(count==0) **then**
7. **for** each instruction Kins after Ins **do**
8. **if** Kins is not an unlock instruction **then**
9. Rprotecte ← dRprotected ∪ Ins;
10. **if** (Ins is Conditional instruction) && (Condition variable f!=null) **then**
11. IFprotected_read ← IFprotected_read ∪ {f};
12. **end if**
13. **else**
14. count++;
15. **end if**
16. **end for**
17. **if**(Ins is a writelock instruction) &&(count==1) **then**
18. **for** each instruction Kins after Ins **do**
19. **if** Kins is not an unlock instruction **then**
20. Wprotected ← Wprotected ∪ Ins;
21. sideEffectAnalysis(Kins, limit, Fprotected_write, Fprotected_read);
22. **end if**
23. **end For**
24. **else** (Ins is a readlock instruction) &&(count==1) **then**
25. **for** each instruction Kins after Ins **do**
26. **if** Kins is not an unlock instruction **then**
27. R1protectedR1protected∪ Ins;
28. **end if**
29. **end for**
30. **end if**
31. **end for**
32. **if**(conditional statement and no conditional end statement in Rprotected)&&(no conditional statement in Wprotected) &&(no conditional statement and conditional end statement in R1protected)&&(Fprotected_write ∩ IFprotected_read ≠ ∅) **then**
33. **return** Sign ← M_signature;
34. **else**
35. **return null;**
36. **end if**

are three critical sections, write-read-write) are reordered, it may result in the same memory location being modified consecutively. When the thread acquires the read lock, the critical section code must read the shared variables from main memory, and the data that should have been read may have been overwritten.

Algorithm 4 presents the basic structure of sequential violation validation, which scans each synchronization method and synchronization block in the class, and analyzes the fields and lock objects. First, the synchronization block or synchronization method in the class is traversed (lines 1-2), and the writing field mapping SWprotectedMap and the read field mapping SRprotectedMap are initialized. The keywords are the lock objects belonging to the field write and read respectively. In the critical section, we divide the protected fields into protected read and protected write, namely Fprotected_read and Fprotected_write (lines 6-9). Each instruction performs side-effect analysis (line 10), and after the analysis, the protected read and protected write are mapped to the corresponding lock object (lines 11-14). If the write mappings of different lock object methods have the same variable, and the same variable is also read in the two methods,

Algorithm 4 Sequential Violation Validation

Input: Cla – the target class
Output: SeqVMap – sequential violation method pair set
1. B/Mset ← all synchronization block and synchronization method from Cla;
2. **for** each B/M in B/Mset **then**
3. limit ← 0;
4. SWprotectedMap ← ∅, SRprotectedMap ← ∅;
5. Instructions ← all instruction of B/M;
6. **for** each Ins in Instructions **do**
7. Ins.lock ← Ins' protected lock
8. SWprotectedMap(Ins.lock) ← ∅, SRprotectedMap(Ins.lock) ← ∅;
9. Fprotected_write ← ∅, Fprotected_read ← ∅;
10. sideEffectAnalysis(Ins, limit, Fprotected_write, Fprotected_read);
11. SWprotectedMap(Ins.lock)←SWprotectedMap(Ins.lock)∪ Fprotected_write;
12. SRprotectedMap(Ins.lock)← SRprotectedMap(Ins.lock ∪ Fprotected_read;
13. SWprotectedMap ← SWprotectedMap ∪ (Ins.lock,SWprotectedMap(Ins.lock));
14. SRprotectedMap ← SRprotectedMap ∪ (Ins.lock, SRprotectedMap(Ins.lock));
15. **end for**
16. **end for**
17. **for** each M_i/B_i in all M/Bset **then**
18. **for** each M_j/B_j after M_i/B_i **then**
19. M_i/B_iWmap ← M_i/B_i.SWprotectedMap;
20. M_i/B_iRmap ← M_i/B_i.SRWprotectedMap;
19. M_j/B_jWmap ← M_j/B_j.SWprotectedMap;
20. M_j/B_jRmap ← M_j/B_j.SRWprotectedMap;
21. **if** M_i/B_iWmap and M_j/B_jWmap have the same variables **then**
22. **if** M_i/B_iRmap or M_j/B_jRmap have the same variables **then**
23. SeqVMap ← SeqVMap ∪ (M_i/B_i, M_j/B_j);
24. **end if**
25. **end if**
26. **end for**
27. **end for**

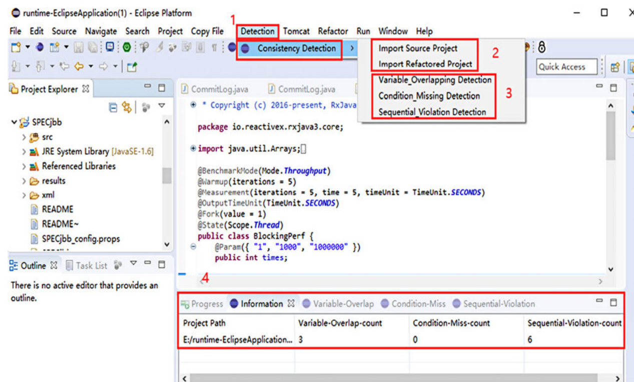

FIGURE 5. Consistency validation tool interface.

it is determined that a sequential violation may occur(lines 17-27).

VI. IMPLEMENTION

We implement our consistency validation tool as an Eclipse plugin. This experiment uses the WALA tool for code analysis. The interface of the validation tool is shown in Figure 5.

The user needs to select the source program of the test and the program after fine-grained lock refactoring as input, and then click the Detection button in the menu bar to start the plug-in, and select the type of validation that needs to be tested. The result will display the project path and the number of categories that caused the synchronization behavior

TABLE 1. Benchmarks and their configuration.

Benchmarks	Before refactoring				After refactoring			
	version	Sync_B	Sync_M	L_Down	L_decomposition	ReadLocks	WriteLocks	refactoring times(s)
HSQLDB	2.4.1	71	613	6	39	109	530	18
Cassandra	3.11.4	13	226	2	24	39	174	73
SPECjbb	1.01	22	168	1	2	58	129	24
JGroups	4.1.5	41	138	5	33	28	113	7
Xalan	2.7.2	31	51	2	5	19	56	19
Fop	2.3	7	25	2	0	9	21	15
RxJava	2.2.13	20	8	0	1	8	19	8
Freedomotic	5.6.0	6	15	2	1	2	16	5
Antlr	4.7.2	13	3	2	5	1	8	5
MINA	2.1.3	3	9	0	3	1	8	2

TABLE 2. Experimental results of validation tools.

Benchmarks	Variable_Overlap	Condition_Miss	Sequential_violation	inconsistencies	consistencies	Validation times(s)
HSQLDB	5	3	17	25	659	384
Cassandra	0	1	9	10	229	371
SPECjbb2005	3	0	6	9	161	127
JGroups	5	1	3	9	170	134
Xalan	2	1	1	4	78	137
Fop	0	0	0	0	32	89
RxJava	0	0	0	0	28	27
Freedomotic	0	1	0	1	20	62
Antlr	0	0	0	0	21	39
MINA	0	0	2	2	10	61
Total	15	7	38	60	1408	1431

TABLE 3. SPECjbb's consistency validation results.

Check category	Quantity	Inconsistent objects	
		Class label	Method
Variable overlap	3	Spec.jbb.validity .syncTest	syncMethod(I)I
			syncMethod2(I)I
		Spec.jbb.TimerData	updateTPMC(J)D
Sequential violation	6	spec.jbb.District	(removeFirstNewOrder(),addNewOrder(Lspec/jbb/NewOrder;)V)
			(removeFirstNewOrder(),removeOldNewOrders(I)V)
			(removeNewOrder(Ljava/lang/Object;),addNewOrder(Lspec/jbb/NewOrder;)V
			(removeNewOrder(Ljava/lang/Object;),removeOldNewOrders(I)V)
			(removeOldestOrder()V,removeOldOrders(I)V)
			(removeOldestOrder()V,removeOldNewOrders(I)V)

change in the display column Information after the execution is completed, and the corresponding method names and classes of inconsistent types are presented to the user in the Variable-Overlap, Condition-Miss, and Sequential-Violation columns.

VII. EVALUATION

This section conducts an experimental evaluation of the proposed tools. First, the experimental configuration and selected test programs are introduced, and then the experimental results are analyzed.

```
1.   public synchronized double updateTPMC(long elapsed_time) {
2.   double temp;
3   .temp=(double)getTransactionCount
(Transaction.new_order) / ((double) elapsed_time / 1000.d) * 60.d;
4.      tpmc += temp;
5.      return tpmc;
6. }
```

(a)Before refactoring

```
1. public double updateTPMC(long elapsed_time) {
2.  tlock.writeLock().lock();
3.  try {
4.  double temp;
5.  temp=(double)getTransactionCount(Transaction.new_
        order) / ((double) elapsed_time / 1000.d) * 60.d;
6.        tpmc += temp;
7.   } finally {
8.        tlock.writeLock().unlock();
9.  }
10.  tlock.readLock().lock();
11.  try {
12.        return tpmc;
13.  } finally {
14.        tlock.readLock().unlock();
15.  }
16.}
```

(b)After refactoring

FIGURE 6. Discovery of variable overlap.

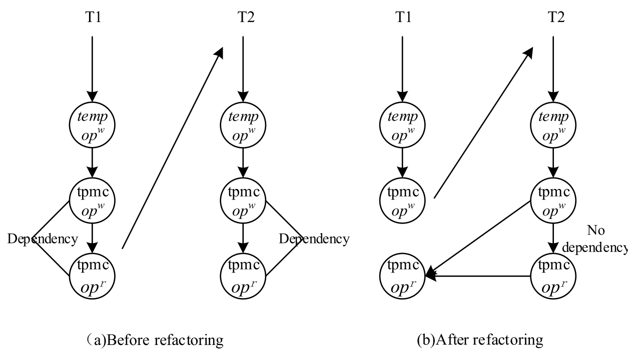

FIGURE 7. Thread execution path diagram.

A. EXPERIMENTAL SETUP

All experiments are done on HP Z240 workstation with 3.6GHz Intel Core i7-7700 processor and 8GB RAM. The workstation runs Windows 10 and has Eclipse 4.12. 0, JDK 1.8.0_221, and WALA 1.5.2 installed.

B. BENMARKS

Ten actual applications were used to evaluate the effectiveness of the proposed validation tool. First, Refactoring operations were performed on these applications by FineLock, the programs before and after refactoring were used as the check objects. These applications include HSQLDB [27], Cassandra [28], SPECjbb2005 [29], JGroups [30], Xalan [31], Fop [32], RxJava [33], Freedomotic [34], Antlr [35], and MINA [35]. HSQLDB is an open-source Java database. Cassandra is an open-source distributed NoSQL database system from Apache. SPECjbb2005 is a Java application server test program. JGroups is a toolkit for reliable messaging. It can be used to create clusters whose nodes can send messages to each other. Xalan and Fop are XSLT transformation processors and formatted object processors from Apache, respectively. RxJava is Netflix's library for composing asynchronous, event-based programs using observable sequences on the Java VM. Freedomotic is an open source, flexible and secure Internet of Things (IoT) development framework. Antlr is a parser generator, and MINA is Apache's web application framework. The version information of these programs, the number of synchronization methods (Sync_B) and synchronization blocks (Sync_M), the number of refactoring operations (lock downgrade, lock decomposition, read lock, write lock) and refatoring times are presented in Table 1.

C. RESULT AND ANALYSIS

In the experiment, the tool was used to check the consistency of the ten benchmarks, and some cases were selected for the results to be displayed.

1) RESULT

After checking the consistency of the above-mentioned benchmark programs, we conducted category statistics on the causes of inconsistent synchronization behavior before and after refactoring in terms of variable overlap due to statement dependencies, competing relationships, and sequential thread execution, and the results are shown in Table 2. It can be seen from the experimental results that there are 60 inconsistencies in the benchmarks. The number of overlapping variables is 15, mainly distributed in HSQLDB, SPECjbb, Xalan and JGroups. The number of missing conditions is 7. In RxJava and MINA, there are no missing condition inconsistencies because the source program contains fewer built-in monitor objects and the refactoring does not perform lock downgrading. The number of sequential violations is 38, and the number of changes in synchronization behavior detected is 17 because the number of refactorings converted to lock decomposition mode in the HSQLDB test program is high; the number of lock decomposition and lock degradation in Fop, RxJava, Antlr and MINA is low, so the number of inconsistencies is low or even absent.

The total time consumed by the 10 test programs is 1431 seconds, and the average time consumed by each program is 143.1 seconds, as shown in Table 2. There are

```
1.private static void registerServer(Notified server, Database db) {
2.   synchronized (serverMap) {
3.     if (!serverMap.containsKey(server)) {
4.       serverMap.put(server, new HashSet());
5.     }
6. HashSet databases = (HashSet) serverMap.get(server);
7.     databases.add(db);
8.   }
9.}
```

(a)Before refactoring

```
1.private static void registerServer(Notified server, Database db) {
2.    stlock.readLock().lock();
3.        if (!serverMap.containsKey(server)) {
4.            stlock.readLock().unlock();
5.            stlock.writeLock().lock();
6.            try{
7.              serverMap.put(server, new HashSet());
8.                stlock.readLock().lock();
9.            }finally{
10.               stlock.writeLock().unlock();
11.           }
12.       }
13.       try{
14.               HashSet databases = (HashSet) serverMap.get(server);
15.               databases.add(db);
16.           }finally{
17.               stlock.readLock().unlock();
18.           }
19.   }
```

(b)After refactoring

FIGURE 8. **Conditional missing test result.**

more synchronization methods and synchronization blocks in HSQLDB, there are 684 of them, and the test time is 384 seconds. Due to the relatively large scale of Cassandra, the time spent for traversal analysis is long, although it is only 10 inconsistencies tested, and the test time is 371 seconds. SPECjbb2005, JGroups and Xalan take 127 seconds, 134 seconds and 137 seconds; RxJava, Freedomotic, Antlr and MINA are relatively small programs, taking about 30 to 60 seconds. For the FOP test program, no inconsistent synchronization behavior was detected, but it also took 89 seconds. By analyzing the validation time of these programs, we found that the validation tool time consumption was mainly used for static analysis of the program, and the larger the program, the longer the static analysis time, causing the total validation time to be longer. Although our validation time did not achieve particularly good results, but the manual validation method will spend a lot of time in the search for code, while the proposed validation tool can automatically complete the inspection, greatly reducing the time consuming.

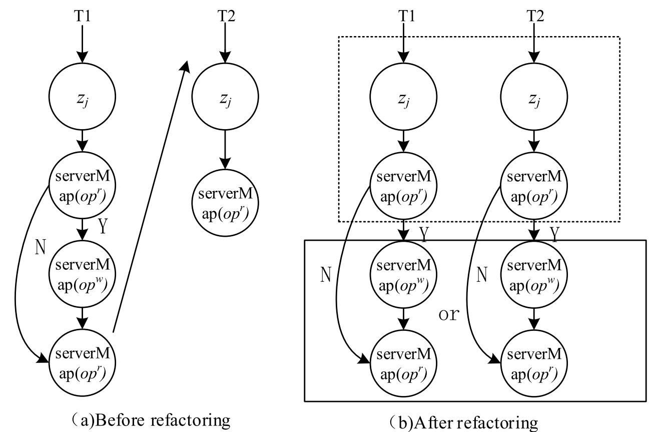

FIGURE 9. **Thread execution path diagram.**

2) CASE STUDY

For the presentation of the found inconsistent objects, take SPECjbb as an example, as shown in Table 3. In the table, the class label lists the paths of the classes where the methods with inconsistent synchronization behaviors exit, and the method lists the method signatures that violate the consistency rules. We have manually checked the reported inconsistent methods, and the synchronization behavior of the listed methods has changed after refactoring. Examination of the TimerData class in the project shows that the method updateTPMC() in the class has overlapping variables. The dependency statement related to the variable tpmc in this class is split into two critical areas due to fine-grained locking, which results in an error. In this project, no missing conditions were detected because the refactoring lock degradation operations were relatively few and almost always refactored correctly. From the test results, the changes in synchronization behavior due to sequential violations are mainly distributed in the District class. This class is mainly used to store the modified user information and zone adjustment.

Our selected program segment from SPECjbb2005 program, Figure 6 shows the refactoring that splits a critical area in Figure 6(a) into critical areas locked by a write lock (lines 2-9) and a read lock (lines 10-15) respectively in Figure 6(b).

```
1.   public class CommitLog
          implements CommitLogMBean{
2.   private boolean started = false;
3.   public synchronized CommitLog start(){
4.       if (started)
5.        return this;
6.        try{
7.            segmentManager.start();
8.            executor.start();
9.            started = true;
10.       } catch (Throwable t){
11.           started = false;
12.           throw t;
13.       }
14.       return this;
15.     }
16. public   synchronized void
              shutdownBlocking() {
17.     if (!started)
18.       return;
19.       started = false;
20.       executor.shutdown();
21.       executor.awaitTermination();
22.       segmentManager.shutdown();
23.       segmentManager.awaitTermination();
24.   }
25.}
```

(a)Before refactoring

```
1.public class CommitLog
      implements CommitLogMBean{
2.private boolean started = false;
3.public CommitLog start(){
4.     nulock.readLock().lock();
5.     try{
6.          if (started)
7.          return this;
8.      }finally{
9.       nulock.readLock().unlock();
10.    }
11.    nulock.writeLock().lock();
12.    try{
13.      try{
14.        segmentManager.start();
15.             executor.start();
16.             started = true;
17.         }catch (Throwable t){
18.         started = false;
19.         throw t;
20.         }
21.       return this;
22.     }finally{
23.       nulock.writeLock().unlock();
24.    }
25.}
26.public void shutdownBlocking(){
27.      tlock.readLock().lock();
28.      try{
29.           if (!started)
30.           return;
31.       }finally{
32.           tlock.readLock().unlock();
33.       }
34.       tlock.writeLock().lock();
35.     try{
36.         started = false;
37.         executor.shutdown();
38.         executor.awaitTermination();
39.     segmentManager.shutdown();
40.segmentManager.awaitTermination();
41.     }finally{
42.       tlock.writeLock().unlock();
43.       }
44.    }
45. }
```

(b)After refactoring

FIGURE 10. Perform sequential test result.

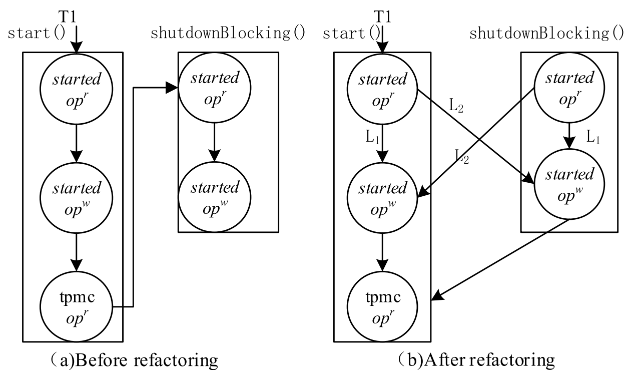

FIGURE 11. Thread execution path diagram.

According to the definition and rule 1 in Section 3, there is a critical area C_i locked by synchronized in the method updateTPMC before refactoring, statement 4 performs a write operation op_{i4}^{w} on the tpmc variable and statement 5 performs a read operation op_{i5}^{r} on it, and there is a data dependency between the two statements. After refactoring, C_{i1} and C_{i2} are split into two critical areas locked by the read/write lock tlock and op_{i4}^{w} and op_{i5}^{r} are distributed to different critical areas, and this splitting operation destroys the dependency relationship.

If there are 2 threads executing this code at the same time, the execution path of the threads is shown in Figure 7. In the source program, since the synchronized modification ensures that the updateTPMC() method is accessed by only one thread at the same time, thread T1 will execute all the statements within the method body before T2 is executed, and this process does not affect the reading of the tpmc value. The refactored execution sequence is shown in Figure 7(b). After executing op_{i4}^{w} on the tpmc variable, T1 releases the write lock, and if T2 acquires the write lock at this time and performs the write operation on the variable, it will certainly affect T1's access to the value of tpmc, which is inconsistent with the original behavior of the program, and we express this phenomenon as variable overlap.

From the analysis of Figure 6 and Figure 7, it can be seen that such cases are consistent with variable overlap verification. According to Algorithm 2, the analysis of the side-effects of instructions within the critical area will result in the read and write mapping WprotectedMap, RprotectedMap about temp and tpmc, which combined with Rule 2 can determine that there is a data dependency between statements 4 and 5 before refactoring and fails to maintain this dependency after refactoring.

Figure 8 shows the case of missing conditions with HSQLDB as an example. The method registerServer() contains the synchronization block with the monitor object serverMap before refactoring, and after fine-grained lock refactoring, the lock downgrading operation is performed

on this synchronization block. The Figure 8(b) shows that the conditional statements and statement blocks related to serverMap are split into two critical section.

If two threads T1 and T2 execute the method, after refactoring, there will be two threads that both read the data first and store it in the thread's own buffer, as shown in Figure 9. If T1 executes the state determination of serverMap first and modifies the state amount after it is satisfied, the judgment of T2 may fail after the execution is finished and affect the subsequent execution. However, before refactoring, T1 executes all the operations and updates the memory before T2 reads the state volume, so it can be seen that such refactoring will cause the condition missing.

From the analysis of Figure 8 and Figure 9, it is clear that such cases are consistent with conditional missing verification, and according to Algorithm 3 and Rule 3, the refactored conditional judgment instruction and the end instruction are distributed to different critical areas, which are prone to competing conditions when accessed by threads.

Taking the CommitLog class examined in Cassandra as an example, Figure 10 illustrates the violation of sequential consistency. As can be seen in Figure 10(a), the method start() and the method shutdownBlocking() both perform judgmental operations on the started variable (line 4-17), and there are data modification statements related to started. After refactoring with the FineLock fine-grained lock refactoring tool, as shown in Figure 10(b). Both methods are refactored to perform lock decomposition operations, and the method start() is decomposed into read locks (lines 4-9) and write locks (lines 11-23) with the lock object nulock, and the method shutdownBlocking() is decomposed into read and write locks with the lock object tlock.

The execution path is shown in Figure 11, as the locking objects before refactoring are Object to ensure the execution order of the method. But after the refactoring, it may cause the data of the start variable to be read in the start() method and then written in the shutdownBlocking() method started, thus causing the subsequent read of started to be wrong, which obviously breaks the original execution order of the program.

From the analysis of Figure 10 and Figure 11, it can be seen that such cases are consistent with sequential violation verification. According to Algorithm 4, by collecting the lock objects of synchronous methods or synchronous blocks, it is judged that the monitor objects of the two methods are consistent before refactoring, and after refactoring, they are locked by different lock objects nulock and tlock, and the threads are prone to inconsistent sequential consistency when accessing with the pre-refactoring.

D. LIMITATIONS

Static analysis is primarily an analysis performed without running the program, while dynamic analysis is a record of function calls as the program actually runs. While dynamic analysis provides access to more call information, such as the order and number of calls, the presence of branching statements in the program may result in certain statements not being executed and not being recorded in the call graph. Dynamic analysis can give a clearer picture of the calls and the execution of threads, and using a combination of dynamic and static analysis to accomplish consistency checking deserves further study.

VIII. CONCLUSION

This paper proposes a refactoring consistency validation method for fine-grained locks. It uses WALA to generate intermediate code to analyze the three behavioral changes caused by the existing refactoring engine: variable overlap, conditional absence, and sequential violation. Then we summarize the verification rules. According to the proposed rules, the variable overlapping validation, condition missing validation, and execution sequence validation are designed to verify the consistency before and after refactoring through call graph analysis, alias analysis, and side-effect analysis.

The validation tool was implemented in the form of an Eclipse plug-in, and the effectiveness of our tool was validated using the fine-grained lock refactoring programs of ten projects including HSQLDB, Cassandra, and Xalan. Experimental results show that the consistency validation tool can effectively check the three concurrency problems mentioned in the paper caused by refactoring. In our future work, we will explore more concurrency issues caused by fine-grained lock refactoring. Moreover, we will use more practical applications to verify the validation tool.

REFERENCES

[1] Y. Zhang, S. Shao, J. Zhai, and S. Ma, "FineLock: Automatically refactoring coarse-grained locks into fine-grained locks," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, New York, NY, USA, Jul. 2020, pp. 565–568.

[2] E. L. G. Alves, M. Song, and M. Kim, "RefDistiller: A refactoring aware code review tool for inspecting manual refactoring edits," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Hong Kong, Nov. 2014, pp. 751–754.

[3] J. Brant and F. Steimann, "Refactoring tools are trustworthy enough and trust must be earned," *IEEE Softw.*, vol. 32, no. 6, pp. 80–83, Nov./Dec. 2015, doi: 10.1109/MS.2015.145.

[4] D. Giebas and R. Wojszczyk, "Atomicity violation in multithreaded applications and its detection in static code analysis process," *Appl. Sci.*, vol. 10, no. 22, p. 8005, Nov. 2020.

[5] C. Zhang, "FlexSync: An aspect-oriented approach to Java synchronization," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, May 2009, pp. 375–385, doi: 10.1109/ICSE.2009.5070537.

[6] K. Maruyama, S. Hayashi, N. Yoshida, and E. Choi, "Frame-based behavior preservation in refactoring," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2017, pp. 573–574, doi: 10.1109/SANER.2017.7884683.

[7] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 147–162, Feb. 2013, doi: 10.1109/TSE.2012.19.

[8] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making program refactoring safer," *IEEE Softw.*, vol. 27, no. 4, pp. 52–57, Jul./Aug. 2010, doi: 10.1109/MS.2010.63.

[9] M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, and P. Borba, "Making refactoring safer through impact analysis," *Sci. Comput. Program.*, vol. 93, no. 11, pp. 39–64, Nov. 2014.

[10] I. P. S. C. Silva, E. L. G. Alves, and W. L. Andrade, "Analyzing automatic test generation tools for refactoring validation," in *Proc. IEEE/ACM 12th Int. Workshop Autom. Softw. Test. (AST)*, May 2017, pp. 38–44, doi: 10.1109/AST.2017.9.

[11] T.-H. Dao, T.-B. Trinh, and N.-T. Truong, "A tool support for checking consistency in model refactoring," in *Proc. 9th Int. Conf. Knowl. Syst. Eng. (KSE)*, Oct. 2017, pp. 100–105, doi: 10.1109/KSE.2017.8119442.

[12] F. F. Silva, E. Borel, E. Lopes, and L. G. P. Murta, "Towards a difference detection algorithm aware of refactoring-related changes," in *Proc. Brazilian Symp. Softw. Eng.*, Sep. 2014, pp. 111–120, doi: 10.1109/SBES.2014.21.

[13] E. A. AlOmar, M. W. Mkaouer, C. Newman, and A. Ouni, "On preserving the behavior in software refactoring: A systematic mapping study," *Inf. Softw. Technol.*, vol. 140, Dec. 2021, Art. no. 106675.

[14] Y. Zhang, S. X. Sun, and D. W. Zhang, "Consistency detection method for concurrent code refactoring," *J. Hebei Normal Univ.*, vol. 44, no. 3, pp. 22–30, 2020.

[15] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Refactoring Java programs for flexible locking," in *Proc. 33rd Int. Conf. Softw. Eng.*, May 2011, pp. 71–80, doi: 10.1145/1985793.1985804.

[16] B. Tao and J. Qian, "Refactoring Java concurrent programs based on synchronization requirement analysis," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Sep./Oct. 2014, pp. 361–370, doi: 10.1109/ICSME.2014.58.

[17] T. Yu and M. Pradel, "SyncProf: Detecting, localizing, and optimizing synchronization bottlenecks," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, Saarbrücken, Germany, Jul. 2016, pp. 389–400.

[18] Y. Zhang, S. Shao, M. Ji, J. Qiu, Z. Tian, X. Du, and M. Guizani, "An automated refactoring approach to improve IoT software quality," *Appl. Sci.*, vol. 10, no. 1, p. 413, Jan. 2020, doi: 10.3390/app10010413.

[19] N. Ubayashi, J. Piao, S. Shinotsuka, and T. Tamai, "Contract-based verification for aspect-oriented refactoring," in *Proc. 1st Int. Conf. Softw. Test., Verification, Validation*, Apr. 2008, pp. 180–189.

[20] X. Yin, J. Knight, and W. Weimer, "Exploiting refactoring in formal verification," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2009, pp. 53–62.

[21] A. Garrido and J. Meseguer, "Formal specification and verification of Java refactorings," in *Proc. 6th IEEE Int. Workshop Source Code Anal. Manipulation*, Sep. 2006, pp. 165–174.

[22] M. Abadi, S. Keidar-Barner, D. Pidan, and T. Veksler, "Verifying parallel code after refactoring using equivalence checking," *Int. J. Parallel Program.*, vol. 47, no. 1, pp. 59–73, Feb. 2019.

[23] P. Hofer, D. Gnedt, A. Schörgenhumer, and H. Mössenböck, "Efficient tracing and versatile analysis of lock contention in Java applications on the virtual machine level," in *Proc. 7th ACM/SPEC Int. Conf. Perform. Eng.*, Delft, The Netherlands, Mar. 2016, pp. 263–274.

[24] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak, and F. Tip, "Correct refactoring of concurrent Java code," in *Proc. 24th Eur. Conf. Object-Oriented Program.*, Maribor, Slovenia, 2010, pp. 225–249.

[25] IBM. *T.J. Watson Libraries for Analysis (WALA)*. [Online]. Available: http://wala.sourceforge.net

[26] *HSQLDB*. Accessed: Dec. 5, 2020. [Online]. Available:http://hsqldb.org/

[27] *Cassandra*. Accessed: Dec. 5, 2020. [Online]. Available: https://cassandra.apache.org/

[28] *SPECjbb2005*. Accessed: Dec. 5, 2020. [Online]. Available: https://www.spec.org/jbb2005/

[29] *JGroups*. Accessed: Dec. 5, 2020. [Online]. Available: http://www.jgroups.org/

[30] *Xalan*. Accessed: Dec. 5, 2020. [Online]. Available: http://xalan.apache.org/xalan-j/

[31] *Fop*. Accessed: Dec. 5, 2020. [Online]. Available: https://xmlgraphics.apache.org/fop/

[32] *RxJava*. Accessed: Dec. 5, 2020. [Online]. Available: http://reactivex.io/

[33] *Freedomotic*. Accessed: Aug. 15, 2020. [Online]. Available: https://www.freedomotic-iot.com

[34] *ANTLR*. Accessed: Dec. 5, 2020. [Online]. Available: https://www.antlr.org/

[35] *MINA*. Accessed: Dec. 5, 2020. [Online]. Available: http://mina.apache.org/

YANG ZHANG received the Ph.D. degree from the School of Computer, Beijing Institute of Technology. He was a Visiting Scholar at Purdue University, in 2017. He is currently an Associate Professor with the School of Information Science and Engineering, Hebei University of Science and Technology. His research interests include parallel programming model and software refactoring for parallelism.

CHUNXIA LI is currently pursuing the master's degree with Hebei University of Science and Technology. Her research interests include parallel programming and software refactoring for parallelism.

YU BAI is currently a Lecturer with the School of Information Science and Engineering, Hebei University of Science and Technology. His research interests include information physical systems (CPS), synchronous systems, deep learning, model-based system design, and formal methods.