Static analysis for enforcing intra-thread consistent locks in the migration of a legacy system

Mariano Ceccato, Paolo Tonella
Fondazione Bruno Kessler—IRST
Trento, Italy
{ceccato, tonella}@fbk.eu

***Abstract*—Often, legacy data management systems provide no native support to transactions. Programmers protect data from concurrent access by adopting commonly agreed patterns, relying on low level concurrency primitives, such as semaphores. In such cases, consistent data access is granted only if all code components are compliant with the adopted mutual exclusion patterns.**

When migrating legacy systems to modern data management systems, the ad hoc mechanisms for data protection must be replaced with modern constructs for transaction management. In such cases, a literal translation may expose problems and bugs, which were originally masked by the specific implementation and patterns in use.

In this paper, we propose a static flow analysis that determines the existence of potentially incompatible locks within the same thread, which require specific code re-engineering before migrating to a modern data management system. We report the results obtained on a concrete instance of this problem.

Keywords: Legacy systems, flow analysis, code migration.

I. Introduction

Legacy systems are built on top of programming frameworks and libraries that often lack native support to transactional access to persistent data. Often, they rely on permissive and low level lock mechanisms (e.g., semaphores). As a consequence, programmers are forced to develop their own access control mechanisms. A consequence of an ad-hoc lock mechanism is that the responsibility of locking/unlocking resources can be scattered among many functions that participate in a program and could be incompatible with modern concurrency control primitives.

For example, in Figure 1, after acquiring the lock on a shared resource (e.g., by means of a semaphore), the caller passes the control to the called library function, that tries to acquire a lock on the same resource. Even if the second lock should clearly fail, in legacy systems concurrency managers often handle locks on a per-thread basis, hence allowing the second lock. In a modern (stricter) transaction management environment, the same code would not work, because the callee's lock would always fail.

The problem in the example in Figure 1 is that the library function needs exclusive access to some data, but there is no way in the legacy language to stipulate a contract with the callers which explicitly declares who is in charge of acquiring the lock and of creating the protected data access context. Hence, the library creates its own protected context. On the other side, the caller is also creating a protected context, since it may also need to protect some data from concurrent

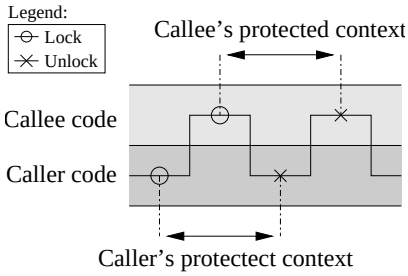

Fig. 1. Interleaving of locks within a thread.

modifications. If the two locks are taken on the same record, which acts as a semaphore for the computation to be protected, we have a problem during migration, in that a double lock on the same record is not permitted. In addition, there may be also a problem in the original system, which might have gone unobserved just by chance. In fact, one of the two components holding a lock (e.g., the caller) may decide to close the protected context, by releasing the lock, before the other component (the callee) is done with the protected computation. In such a case, nothing is protecting the persistent data from concurrent modification in the interval between the first and the second lock release, which, in our example, may bring the callee to an inconsistent state. At the core of the problem is the lack of a clear, explicit attribution of responsibility on the creation and management of a protected data manipulation context.

The problems highlighted with reference to the example in Figure 1 become dramatic when legacy systems are migrated to a modern platform, supporting persistent data management with native access control. The manually implemented, oftentimes very permissive lock/unlock mechanisms may give raise to undesired behaviors, like errors, exceptions, deadlocks or inconsistent/incompatible data access. In this work, we investigate the use of static flow analysis to explicitly determine incompatible lock acquisitions. These cases must be refactored before migrating to a modern platform in order to avoid the occurrence of deadlocks or errors. We report the results obtained on a large (8 millions LOC) legacy system – a complete bank management system – that we are currently migrating from a proprietary language (BAL, Business Application Language) to Java.

In this work, the focus is not the problem of concurrent lock acquisition performed by multiple, parallel threads. Such a problem has been broadly investigated in the past and authoritative solutions have been proposed [1], [2]. Instead, we are interested in detecting and resolving lock inconsistencies that originate from serial execution within a single thread, when nested or overlapping protected contexts might interfere (see Figure 1).

The paper is organized as follows: Section II explains lock analysis and provides the details of the algorithm we used to determine inconsistent locks statically. Section III describes the results obtained when applying our lock analysis to the legacy system we are migrating to Java. Section IV provides our preliminary catalog of refactorings that can be used to solve the detected inconsistent lock problems. Related works are commented in Section V, while the last section is devoted to conclusions and future work.

II. Lock analysis

Let us consider a legacy programming language and environment which does not support the explicit attribution of lock responsibilities. As a consequence, different components (e.g., library code vs. code for the main program) may realize a local lock policy, which appears to be completely consistent within the component, but may originate inconsistent locks due to the interaction with other components, when these are not aware of the lock pattern implemented elsewhere.

In such cases, inconsistencies arise when locks escape the component that generated them and propagate to other components on the same thread. If another source of lock in the other components operates on the same record that was locked externally, problems (e.g., errors, exceptions or deadlocks) are potentially originated when a stricter lock policy is adopted. Moreover, even the original program might experience problems at run time, since such inconsistencies may originate computation intervals in which data access protection is not ensured.

In the following, we will no longer distinguish between the source of the lock and the component containing it, since for our purposes (detecting inconsistencies) knowledge about the source container is sufficient. So, by a lock source s_1 we mean a component, library or well defined code portion which implements a consistent, local lock policy. Another simplification that we make is about the locked record. Since we are interested in static lock analysis and static analysis cannot distinguish different records accessed from the same table, we will not distinguish between table and record. When talking about lock on a table T we do not mean that the entire table is locked. Rather we mean that one record from the table is locked, but statically we do not know exactly which one, so we over-approximate the lock with the entire table.

A. Inconsistent locks

A lock request L_1 on table T performed by lock source s_1 is inconsistent with the lock request L_2 performed by lock source s_2 on the same table T if the two sources of lock are different ($s_1 \neq s_2$) and there exists an interprocedural path in the interprocedural control flow graph of the program from the statement acquiring the first lock to the statement acquiring the second lock, such that no intermediate statement in the path releases the lock on the same table.

```
function s1(void){
  lock(T1)
  lock(T2)
  s2()
  unlock(T2)
  unlock(T1)
}
```

```
function s2(void){
  lock(T1)
  lock(T2)
  unlock(T2)
  if (a)
    unlock(T1)
  lock(T3)
}
```

Fig. 2. Example of inconsistent lock.

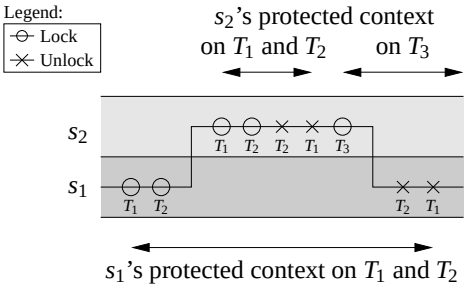

Fig. 3. Lock/unlock performed by s_1 and s_2.

The situation we are interested in is one where a lock L_1 escapes the component s_1 where it is managed consistently and reaches another component s_2 where it is not ensured not to conflict with other locks. In particular, if the latter component acquires a lock on the same table, we have potentially a problem if the first lock can reach the second without being released. Of course, if the two statements are always locking different records, we have no real problem, even though a static analysis would report an inconsistent lock. However, this is an intrinsic limitation of static analysis, which cannot determine the exact record being accessed (this is decided at run time), but can quite precisely determine which table is locked.

Figure 4 shows the interprocedural control flow graph for the program in Figure 2, while Figure 3 shows the scattered lock responsibility for T_1 and T_2. This program includes two lock sources, s_1 and s_2. The inconsistent locks that exist in this program can be split into two categories: (1) inconsistencies due to bottom-up lock propagation; and, (2) inconsistencies due to top-down lock propagation. The two kinds of inconsistent locks depend on whether the lock originates inside nested calls in the call graph and propagates upward, to the calling functions, or whether it is generated at calling functions and it later reaches conflicting locks inside called functions. One example of lock of the first category is *lock(T_1)* inside s_2. This lock reaches the end of s_2, when the *false* branch is taken at the conditional statement *if(a)*. Within the calling function (s_1), this lock collides with another *lock(T_1)* that reaches the call statement $s_2()$ of s_1. One example of lock of the second category is *lock(T_2)* inside s_1 (but the same argument holds for *lock(T_1)* inside s_1), which propagates to s_2 through the

call node $s_2()$ of s_1. Inside s_2 this lock reaches the statement *lock(T_2)*, where it gives raise to an inconsistency.

The lock analysis problem can be formulated as a flow analysis problem, where the flow information being propagated in the interprocedural control flow graph consists of a pair whose first element indicates the table T that was locked by a given statement n and whose second element indicates the lock source s. For brevity, we indicate such pair as $T\langle s\rangle$.

The information generated and killed at each control flow graph node n can then be defined as follows:

$$GEN[n] = \{T\langle s\rangle | statement\ n \in s;\ n\ locks\ T\} \quad (1)$$

$$KILL[n] = \{T\langle s'\rangle | statement\ n\ releases\ T;\ s'\ is\ any\ lock\ source\} \quad (2)$$

The flow propagation is in the forward direction, with union as the meet operator at junction nodes. Flow analysis terminates when the following equations produce the least fixpoint:

$$IN[n] = \bigcup_{p \in\ pred(n)} OUT[p] \quad (3)$$

$$OUT[n] = GEN[n] \cup (IN[n] \setminus KILL[n]) \quad (4)$$

where *pred(n)* indicates the set of nodes that precede immediately n in the control flow graph. Since we are interested in interprocedural flow propagation (lock inconsistencies are intrinsically interprocedural), we split our flow analysis into two steps. First we perform a bottom up propagation, in which we propagate the flow information inside leaf functions in the call graph first and then we proceed backward to the calling functions. In the presence of mutual recursion, we treat an entire strongly connected component as a single unit (pseudo-function) of the call graph. The bottom up propagation results in a summary flow equation for each function in the program. It also reveals inconsistent locks that are generated deeply in the call graph and propagate backward to the upper levels (see next subsection for more details).

Then, we perform a top-down flow propagation in the call graph. During this step, we use the summary information computed in the previous step to account for the effects of function calls on the flow information being propagated. After this step, we can detect inconsistent locks that originate up in the call graph and propagate downward through a sequence of function calls (see subsection after the next one).

At the end of each step we can detect different inconsistent locks, by verifying if the following conditions hold:

Inconsistent lock detection

We have an inconsistent lock when:

$$T\langle s_1\rangle \in IN[n] \quad (5)$$

$$T\langle s_2\rangle \in GEN[n] \quad (6)$$

$$s_1 \neq s_2 \quad (7)$$

	Summary-lock	Summary-unlock
s_2	$T_1\langle s_2\rangle$, $T_3\langle s_2\rangle$	T_2
s_1	$T_3\langle s_2\rangle$	T_1, T_2

TABLE I
BOTTOM-UP SUMMARIES FOR S_1 AND S_2.

B. Bottom-up lock propagation

The bottom-up flow analysis applies intraprocedural flow analysis and computes *summary-out* information for functions. Summary-out information can be divided into *summary-lock* and *summary-unlock*. These two summaries are used on function calls to propagate any lock/unlock due to the call in the caller scope. They also determine which locks are propagated after the call. The first step to compute summary-locks consists of propagating locks according to equations (3) and (4) inside leaf functions (strongly connected components, if mutually recursive functions exist in the program) in the call graph. Summary-lock is then computed as the union of the locks that reach the exit nodes. These locks are the ones that escape the called component and possibly interfere with locks in calling components.

$$summary\ lock(f) = \bigcup_{n=exit\ node\ of\ f} OUT[n] \quad (8)$$

Summary-unlock requires different flow equations, because we want to collect all the unlocks that *for sure* reach *all* the exit nodes. They represent all the locks that are conservatively released when the given function is called, and that must not be propagated after the call, inside the caller. The meet operator is intersection and the new equations are:

$$IN[n] = \bigcap_{p \in\ pred(n)} OUT[p] \quad (9)$$

$$OUT[n] = (IN[n] \cup KILL[n]) \setminus GEN[n] \quad (10)$$

$$summary\ unlock(f) = \bigcap_{n=exit\ node\ of\ f} OUT[n] \quad (11)$$

Considering the example in Figure 2, because of the call in s_1 to s_2, the summary edge for s_2 must be available before considering s_1. In this example we assume that s_1 and s_2 come from different components. Figure 4 shows their control flow graphs and Table I reports the resulting summaries.

Starting from s_2 (right-hand side) we see that the lock $T_2\langle s_2\rangle$ does not reach the exit because it is killed by the subsequent unlock T_2. Whereas, the lock $T_1\langle s_2\rangle$ reaches the exit on the flow that passes through the *false* branch of the if statement, it is not killed by the unlock in the *true* branch. The lock $T_3\langle s_2\rangle$ also reaches the exit node.

The summary-unlock for s_2 contains just T_2, because such unlock is not killed by any other lock on the same table and it reaches the exit node on all the paths. The summary-unlock does not contain T_1, because the unlock holds only on the path

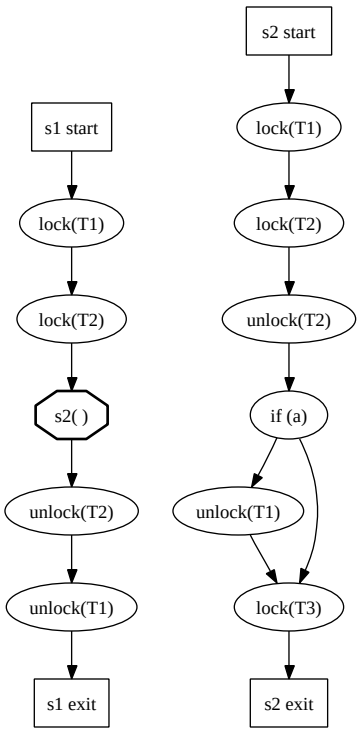

Fig. 4. Control flow graph for s_1 and s_2.

that goes through the *true* branch of the if statement (an unlock must hold on all paths to the exit to be in the summary).

Now we can move to s_1, Figure 4 left-hand side. After acquiring lock $T_1\langle s_1\rangle$ and $T_2\langle s_1\rangle$, summary data available for s_2 can be used in the call point (lock summary contains $T_1\langle s_2\rangle$, $T_3\langle s_2\rangle$ and unlock contains T_2). Here the first inconsistency, I_1, is found where $T_1\langle s_1\rangle$ and $T_1\langle s_2\rangle$ collide after the call. There is a flow where two locks on the same table (i.e., T_1) collide and they are originated on different components (i.e., s_1 and s_2).

Locks $T_2\langle s_1\rangle$, $T_1\langle s_1\rangle$ and $T_1\langle s_2\rangle$ are later killed by unlocks T_1 and T_2. The only lock that reaches the exit of s_1 (and thus is represented in the summary-lock) is $T_3\langle s_2\rangle$. It is acquired by s_2 and never released by both s_2 and s_1. The summary-unlock for s_1 is represented by T_1 and T_2.

C. Top-down lock propagation

The purpose of the top-down propagation is to compute summary-in, locks that hold when the control is passed to a called function. Summary-in contains only summary-locks that are propagated according to equations (3) and (4). Summary-in for a given function is computed as the union of the locks that reaches the calling nodes.

$$summary\ in(f) = \bigcup_{n=call\ to\ f} IN[n] \qquad (12)$$

Using the running example in Figure 2, the top-down ordering associated with the call relation imposes to process s_1 before s_2.

	Summary-in lock
s_1	
s_2	$T_1\langle s_1\rangle$, $T_2\langle s_1\rangle$

TABLE II
TOP-DOWN SUMMARIES FOR S_1 AND S_2.

Name	Locks	Bottom-up	Top-down
I_1	$T_1\langle s_1\rangle$, $T_1\langle s_2\rangle$	X	X
I_2	$T_2\langle s_1\rangle$, $T_2\langle s_2\rangle$		X

TABLE III
INCONSISTENT LOCKS FOR S_1 AND S_2.

Looking at the flow graph in Figure 4 left-hand side, we see that s_1 acquires locks $T_1\langle s_1\rangle$ and $T_2\langle s_1\rangle$, then it calls s_2. This is the only call to s_2, so the summary-in of s_2 contains just these two locks (reported also in Table II). Summary-out of s_2 (already computed in the bottom-up phase) is used here to proceed with the flow analysis of s_1. This allows to find an inconsistency already reported above for the call node $s_2()$, i.e. I_1 containing $T_1\langle s_1\rangle$ and $T_1\langle s_2\rangle$.

The *start* edge of s_2 is assigned the flow value corresponding to its summary-in, i.e. the union of the locks possibly existing when the function is called, in this case $T_1\langle s_1\rangle$ and $T_2\langle s_1\rangle$. On the next edge in the control flow, a lock is acquired ($T_1\langle s_2\rangle$) that collides with a lock in the summary-in ($T_1\langle s_1\rangle$). This inconsistency is I_1 and was already reported earlier, by both the bottom-up and the top-down analysis. In the next edge a second lock is acquired ($T_2\langle s_2\rangle$) that also collides with a summary-in lock ($T_2\langle s_1\rangle$). This second inconsistency I_2 is a new one, which was never reported before.

III. EXPERIMENTAL RESULTS

A. Case study

The proposed flow-analysis for inconsistent lock detection has been applied to a large legacy banking application consisting of 8 millions lines of BAL code. BAL is an acronym for Business Application Language, a BASIC like language that contains unstructured data elements as well as unstructured control statements (e.g., `GOTO`). The migration of the legacy application to a modern programming language (Java) involves a change in the data access libraries. The new library implements a more restrictive locking policy, so a detailed analysis is required to avoid locking errors. More information about the migration project and the problems associated with restructuring the data and the control flow can be found in our previous publications [3], [4].

Persistent data structures are stored on ISAM[1] tables and they are accessed by I/O primitives. BAL provides I/O primitives to open, search, scroll and modify records on ISAM table. These primitives can optionally lock the accessed record so as

[1]ISAM (Indexed Sequential Access Method) is a data file format quite common in old legacy systems.

to open a transactional context. Lock is released by performing a non-locking access on the same record.

On the current release, BAL delegates I/O operation to a C-ISAM library that adopts a permissive lock policy by allowing a thread to acquire multiple locks on the same record. This typically happens when different file handles are used to access the same table. Indeed, different file handles are necessarily used in different components (e.g., main program code and library code), since each component implements its own file handle table. Even if a record is locked, it can be accessed by any called function, both via locking and non-locking primitives, without any lock-violation error. Actually, the lock state of a record is used just as a semaphore for concurrent threads, but it does not carry any intrinsic transactional context semantics within a thread. It is only the programmers' coding style that adds a transactional meaning on top of the available semaphores.

Before migrating the system to Java, the C-ISAM library will be replaced by D-ISAM, since only the latter was ported to Java. The problem is that the D-ISAM library implements a more restrictive lock policy, which does not permit the same thread to acquire multiple locks on the same record. Inconsistent locks have to be identified and solved to avoid run-time lock-errors that could possibly raise an exception and crash the application. However, it should be noticed that inconsistent lock identification is beneficial also for the original BAL code with the C-ISAM library, since such inconsistencies may cause unprotected computation intervals, in which persistent data are exposed to concurrent, uncontrolled modifications. Those are the typical concurrency bugs which are very difficult to expose during testing, since they require very specific interleavings of parallel threads. Revealing them by means of static analysis is thus particularly useful to improve the quality of the code.

Differently from C-ISAM, D-ISAM does not allow a thread to acquire a lock on the same record by means of different file handles, which previously used to happen regularly, when two different components having their own file handle tables were accessing the same record. More precisely, in BAL there are two independent file handle tables, one for the main program and one shared by all libraries. Hence, in the rest of the paper we assume that a component is either (1) the current main program; or, (2) any called library. This means that, in practice, we have an inconsistent lock when:

1) After locking a record, the program invokes a library that tries to acquire a lock on the same record; or
2) The program tries to lock a record already locked by a previously called library, that did not release the lock.

Since static analysis is not able to determine which record is accessed, we conservatively extend the lock to the entire table. Manual investigation is required to understand whether two inconsistent locks are actually colliding (on the same record), or are just false positives (i.e., always different records of the same table).

B. Tools

Of the analyses described in Section II, to date only the bottom-up part was implemented, using Txl [5] and Java. The tool works in the following steps:

1) Preprocessing;
2) Total ordering;
3) Control flow graph extraction; and
4) Summary edges and bottom-up flow analysis.

a) Prerpocessing: A preprocessor, implemented in Txl, is applied to the code (as done in a previous work [3]) in order to remove some syntax ambiguity due to BAL and to apply unique naming to variables. Moreover, all the macros are expanded. Indeed, often data are accessed using macros, in order to avoid the complicated BAL I/O syntax.

b) Total ordering: Call to external functions are fully resolved. Total order given by call relations among functions is used to sort functions so as to be able to apply bottom-up and top-down analysis. Strongly connected components are considered when mutual recursion is present. To identify them, we use a well-known algorithm, working on directed graphs [6].

c) Control flow graph: Programs are split into the composing sub-modules (i.e., functions and segments) and their control flow graph is elaborated using TXL and stored in XML format. This representation contains

- List of formal parameters;
- A node for each statement in the code;
- Entry node and exit nodes;
- Predecessors and successors for every node;
- Lock and unlock with the corresponding physical table name; and
- Fully resolved calls with the list of actual parameters;

d) Summary edges and bottom-up flow analysis: The actual bottom-up analysis proceeds as specified by the total ordering on call-relations, so that summary-out data would always be available when required.

The actual analysis has been implemented in Java using the XOM[2] library to parse the XML representation of the source code. For all the calls in the graph, summary-out lock/unlock data about the called function are read and stored with the corresponding call nodes. Flow analysis is applied twice on the control flow graph. On the first iteration, equations (3) and (4) are used to propagate locks. The second iteration is devoted to unlocks, so equations (9) and (10) are used.

When the fix-point is reached, the control flow graph of the function is visited and *OUT* values are inspected to check for inconsistent locks. Summary-lock and summary-unlock are eventually computed on exit nodes, using (8) and (11) to merge values at the exit nodes. Summaries are then saved for later use, whenever the current function is called.

1) Formal parameters: Sometimes the name of the table on which a lock is taken or released is not known, since it is a formal parameter of the function. In such cases, we apply the

[2]http://www.xom.nu

algorithm described in the previous section with a symbolic table name rather than a concrete one. In this way, we can still conservatively determine summary-outs for functions, but the values in the summary-outs may include symbolic table names. The binding between symbolic and concrete names is deferred to when the summary-out is actually used, i.e., when calling functions are considered. Within a calling function we can either have a concrete table name bound to the symbolic name, or the function may also use one of its formal parameters as the table name. In the latter case, the summary for the calling function will also include a symbolic table name.

2) Sources of imprecision: Although the proposed algorithm is based on a *conservative* static analysis, which gives raise only to false positives, associated with infeasible execution paths, there are a few sources of imprecision which may give raise to false negatives as well.

The first source of imprecision is given by table names. Table names are strings. Usually, constant strings are used or constant strings are assigned to string variables which are then used to acquire or release a lock. However, in principle any string manipulation operation could be used to produce the table name used in a lock acquiring/releasing statement. Our tool currently misses such sources of lock/unlock.

The second source of imprecision is given by logic numbers. Even though BAL programmers tend to prefer the use of macros to acquire/release locks, and macros accept table names as parameters, sometimes they resort to low level BAL I/O primitives, which use logic numbers (similar to file handles), instead of table names. In principle, logic numbers can be computed in any arbitrary way, since they are just short integers. However, the typical pattern consists of an *ASSIGN* BAL statement, which directly associates a logic number with a physical table.

Finally, formal parameters are handled properly only under the assumption that they are directly used, without any string manipulation operation intervening on them. In fact, it is only under this assumption that we can use symbolic table names instead of concrete ones in our analysis. When this prerequisite does not hold, we may be unable to determine the correct table name when the binding between formal and actual parameters is performed.

We cannot be completely sure that the three sources of imprecision mentioned above never occur in the code. However, we performed some preliminary analyses, both by manually inspecting the code and by running some simple scripts that we developed for this purpose, and we found that our assumptions are generally met. We plan to make these checks more rigorous in the future, but with the currently available information we are quite confident that these sources of imprecision have minimum or no impact on the accuracy of the results.

C. Inconsistent locks found

Over the entire banking application (640 programs, 116,953 functions/segments), a total of 1,920 instances of inconsistent locks have been found. This would mean an average of 3 inconsistencies per program and an inconsistency every 61 functions. However, at a deeper analysis the distribution of inconsistencies appears to be not regular. In fact, only 14% of programs (90) contains inconsistencies and, within them, less than 1% of the functions (334). Moreover, among them, the distribution is also not homogeneous. In fact just a few of them contain most inconsistencies and a lot of modules contain few cases.

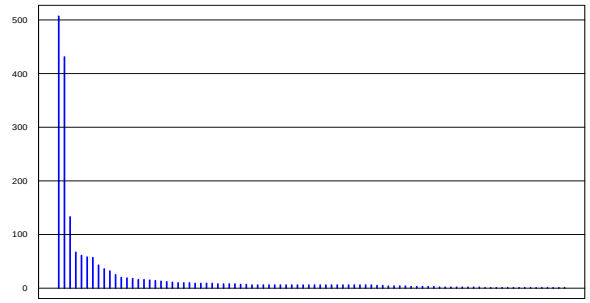

Fig. 5. Distribution of inconsistent locks among BAL programs.

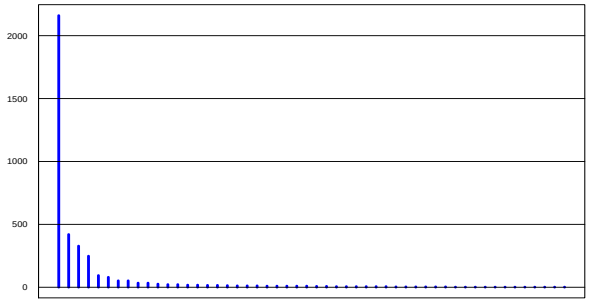

Fig. 6. Distribution of inconsistent locks among ISAM tables.

Figure 5 shows the distribution of inconsistencies among programs. In the worst case we report more than 500 inconsistencies, followed by one case with 431 and one with 133. There are 21 programs with between 100 and 10 inconsistencies. For the remaining 66 programs, we report less than 10 inconsistencies.

For functions, the distribution is similar. In the worst case, we report 290 inconsistencies, followed by a case with 136 inconsistencies. Among the other 41 functions, there are between 100 and 10 inconsistencies. The remaining set of functions (290) present less than 10 inconsistencies.

A similar distribution, shown in Figure 6, holds if the inconsistency rate is evaluated on the affected tables. For the most affected table, more than 2,000 cases are reported. For the second one, this amount is less that one fifth (420). In total only 4 tables have more than 100 inconsistencies, and just 22 tables are between 100 and 10 cases. The other tables (26) have less than 10 inconsistencies.

IV. Re-engineering the data access to solve inconsistent locks

Once inconsistent locks are detected, refactoring can be used to remove the problems and to allow library upgrade (from C-ISAM to D-ISAM, required for the migration to Java). Refactoring strategies are selected manually, case by case.

```
function int f1()
  key1 = "1020304"
  //lock(TABLE_1)
  search_lock(TABLE_1, "A" + key1)
  ...
  f2(key1)
  ...
  //unlock(TABLE_1)
  modify(TABLE_1, "A" + key1)
endf
```

```
function int f2(string key1)
  //lock(TABLE_1)
  search_lock(TABLE_1, "B" + key1)
  ...
  //unlock(TABLE_1)
  modify(TABLE_1, "B" + key1)
endf
```

Fig. 7. Case 1.1: Locks on different keys.

Based on what we observed in the code, a preliminary catalog of possible inconsistency instances and resolution strategies has been distilled. We report it below.

A. Case 1: No re-engineering required

a) Case 1.1: Locks on different keys.: As anticipated earlier, since static analysis cannot tell which record is locked, locks are reported as inconsistent if they apply to the same table. In Figure 7, an example is shown of two locks on table `TABLE_1` by functions from different components, namely `f1` and `f2`. However, manual inspection of the code reveals that locks are on records identified by different keys (in fact, `"A" + key1` $\neq$ `"B" + key1`). So no lock-error would ever happen at run-time, thus this case does not require any fix.

The case depicted in Figure 7 is quite typical in the BAL code we are migrating. In fact, the key prefix is often used as a discriminator for the record type. The same table hosts different kinds of records (i.e., it is a union table), which are distinguished by key prefix. When different code portions operate on different record types, we are sure that different key prefixes will be used to access the common union table, hence no lock conflict can ever occur.

b) Case 1.2: Unlock missing on error.: Since the permissive lock policy does not raise errors in case of inconsistency, an unlock could be missing simply because of a programming error. However, static analysis is not able to judge whether unlock is missing on purpose or by error.

For example, Figure 8 shows an example of missing unlock that generates an inconsistency when this function is called in a locking program. Specifically, in case the locking `search` produces an error (return value $\neq 0$), the function terminates without releasing the lock. However, this is done on purpose by programmers, because the locking `search` does not acquire any lock when it returns an error code. The problem is that our analysis is not able to distinguish between error handling code and normal business logic. This case does not require any fix.

c) Case 1.3: Constrained path.: False positives may also be reported, because of the intrinsic limitations of static analysis. In Figure 9 a case is shown where the lock is selectively

```
function int update_protocol(string data)
  int err = -1

  buffer = data
  //lock(TABLE_1)
  err = search_lock(TABLE_1, buffer)
  if (err != 0) goto &END
  ...
  //unlock(TABLE_1)
  err = modify(TABLE_1, buffer)
&END
  return err
endf
```

Fig. 8. Case 1.2: Unlock missing on error.

```
select mode
  case 1
    //lock(TABLE_1)
    search_lock(TABLE_1, buffer)
  case 2
    //lock(FILE_1)
    search_lock(FILE_1, buffer)
endsel
  ...
select mode
  case 1
    //unlock(TABLE_1)
    modify(TABLE_1, buffer)
  case 2
    //unlock(FILE_1)
    modify(FILE_1, buffer)
endsel
```

Fig. 9. Case 1.3: Constrained path.

released, causing the lock to propagate to the calling context and possibly causing inconsistencies with forthcoming locks. However, because of the business logic of the code, it happens that locks are always released, but our static analysis is not be able to reveal that. In fact the same condition guards a lock and the corresponding unlock, so that they perfectly match.

If manual inspection is not able to determine that locks are always on the different records, a fix is required to avoid a run-time error when adopting a strict lock manager. One of the subsequent refactoring must be applied, based on manual inspection of the source code.

B. Case 2: Reduce protected context to the minimum

The protected context on the caller could be unnecessarily large, for example because unlocks have been relegated to the end of a function body. So statements that occur in the protected context may be not necessarily related to locked resources. The unlock (lock) can be anticipated before (deferred after) these statements, without altering the whole semantics. In Figure 10, the caller protected context has been shrunk by moving the unlock before the invocation to the callee. In this way the lock on the callee will not fail because of the caller lock.

C. Case 3: Postpone or anticipate nested context

Manual inspection of the code could reveal that the invocation of the callee can be safely moved without altering the whole semantics of the program. In this case the call could be deferred (or anticipated) after (before) the caller protected context so as to avoid the interference between caller and

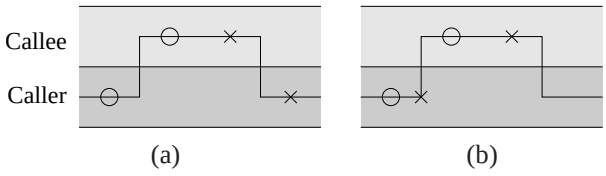

Fig. 10. Case 2: Reduce protected context to the minimum, (a) before and (b) after the refactoring.

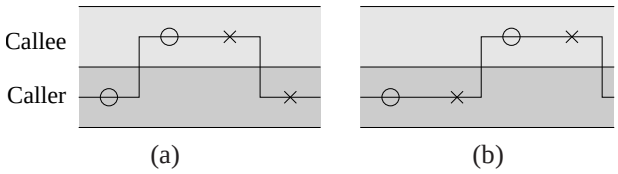

Fig. 11. Case 3: Postpone or anticipate nested context, (a) before and (b) after the refactoring.

callee. In Figure 11, the invocation has been moved after the resource is unlocked by the caller. The difference with respect to Case 2 is that in Figure 11 we are not moving just the unlock statement: we are moving an entire sub-computation, which originally was after the callee, before the call point. This is possible only if the move does not affect the overall execution semantics. Otherwise, a deeper code refactoring may be necessary to ensure that the global effect of the execution of Figure 11 (b) is the same as in Figure 11 (a).

D. Case 4: Inline nested context

When neither statements nor locks/unlocks can be moved, an alternative strategy is represented by inlining the code from the protected context of the callee into the protected context of the caller. This refactoring requires substantial manual effort to ensure that the merged protected contexts will not cause any data corruption. It has also the disadvantage of introducing some level of code duplication. In Figure 12, locks and unlocks are removed from the callee, and all its protected context is inlined. After refactoring, the callee contains just non-interfering code.

V. Related works

The problem of migrating a legacy software system to a novel technology has been widely addressed in the literature by different approaches. The different strategies have been classified by Bisbal et al. [7] into (1) redevelopment from scratch; (2) wrapping; and (3) migration. In their view, even the migration strategy requires substantial redevelopment. Our contribution is part of a migration project that belongs to the third class. This paper is part of a bigger project devoted to the migration of a legacy system. In the past, other problems have been addressed to prepare the legacy code for migration, such as GOTO elimination [4] and the recovery of an Object Oriented data model [3].

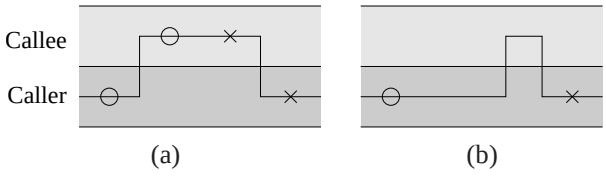

Fig. 12. Case 4: Inline nested context, (a) before and (b) after the refactoring.

Transaction management [1] and concurrency control [2] in the presence of multiple processes are considered consolidated topics. They have been largely studied in the past, both for centralized and distributed database systems. The issue of evaluating properties held by concurrent programs has been largely addressed using model checking [8], Petri nets [9] and data flow analysis [10] [11]. However, the present paper addresses a fundamentally different case. In fact, even if we deal with inconsistent locks, locks are not caused by different threads that concurrently compete in accessing the same resource. Rather, inconsistencies detected by our approach are originated by the same thread when locks are acquired from different components.

Among the mentioned works, the most related ones are by Dwyer et al. [10] and by Naumovich et al. [11]. Dwyer et al. [10] propose an approach to decide whether a particular property is always or never satisfied by a concurrent program. Programs and properties, respectively represented as trace flow graphs and quantified regular expressions, are subject to data flow analysis. The precision of this approach can be tuned so as to allow a trade off between accuracy of the analysis and the required execution time. This approach, initially designed for the Ada rendezvous synchronization model, has been extended by Naumovich et al. [11] to cope with Java specific constructs. Java multi-thread programs are analyzed to spot typical concurrency related problems such as premature join, re-start of already running threads, waiting forever, unnecessary notifications and dead interactions.

The present paper is based on flow analysis [12], a versatile analysis framework for implementing static analysis that propagates user-defined information on the program control flow. It has been widely adopted for a large variety of analyses, for constant propagation and other optimizations implemented in compilers [13], points-to [14] and alias analysis [15], program slicing [16]. More recently flow analysis has been used to detect vulnerabilities in Web applications, such as cross site scripting [17] and SQL-injection [18].

VI. Conclusions

When migrating legacy systems, permissive lock policies often available in the past may be no longer available in more recent technologies. So, migration can be a good opportunity to identify lock problems that went undetected in the past just by chance. In this paper, we address the problem of detecting inconsistent locks. We faced this problem when a stricter lock model was enforced by a library adopted as a step of the migration of a large legacy banking system to Java.

We implemented an interprocedural static analysis based on flow analysis, where lock information is propagated together

with the components where locks originate. Identified inconsistent locks have been analyzed and a catalog of possible refactorings has been defined to solve problematic cases, so as to avoid that inconsistent locks could cause run-time errors when more recent libraries are adopted. Solving the detected inconsistencies is also beneficial for the original legacy code, where problems may have not been exposed just because the precise interleavings which reveal them is hard (but maybe not impossible) to produce. We found that most inconsistent lock problems affect a relatively small number of programs, functions and ISAM tables, hence by refactoring them manually we expect to achieve a major improvement with limited programming effort. Such work is currently ongoing at the company which is performing the migration.

Our future work will be devoted to: (1) completing the implementation and reporting also the problems associated with the top-down propagation of locks; (2) (semi) automatically verifying if the assumptions which make our analysis conservative (no false positives) actually hold (or determine the cases that violate such assumptions); (3) finish the refactoring of the BAL code (this is an ongoing task that our industrial partner is carrying out on its own, based on the output of our analysis).

REFERENCES

[1] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[3] M. Ceccato, T. R. Dean, and P. Tonella, "Recovering structured data types from a legacy data model with overlays," *Information and Software Technology*, vol. 51, no. 10, pp. 1454 – 1468, 2009.

[4] M. Ceccato, P. Tonella, and C. Matteotti, "Goto elimination strategies in the migration of legacy code to java," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*, K. Kontogiannis, C. Tjortjis, and A. Winter, Eds. IEEE Computer Society, April 2008, pp. 53–62.

[5] J. Cordy, "The TXL source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, August 2006.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, third edition*, T. M. Press, Ed., 2009.

[7] J. Bisbal, D. Lawless, B. Wu, and J. Grimson, "Legacy information systems: issues and directions," *Software, IEEE*, vol. 16, no. 5, pp. 103–111, Sep/Oct 1999.

[8] J. C. Corbett, "Constructing compact models of concurrent java programs," in *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 1998, pp. 1–10.

[9] C. Demartini and R. Sisto, "Static analysis of java multithreaded and distributed applications," in *PDSE '98: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 1998, p. 215.

[10] M. B. Dwyer and L. A. Clarke, "Data flow analysis for verifying properties of concurrent programs," *SIGSOFT Softw. Eng. Notes*, vol. 19, no. 5, pp. 62–75, 1994.

[11] G. Naumovich, G. S. Avrunin, and L. A. Clarke, "Data flow analysis for checking properties of concurrent java programs," in *In Proceedings of the 21st International Conference on Software Engineering*, 1999, pp. 399–410.

[12] M. Sharir and A. Pnueli, *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981, ch. Two approaches to interprocedural data flow analysis, pp. 189–233.

[13] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers principles, techniques, and tools*. Reading, MA: Addison-Wesley, 1986.

[14] B. Steensgaard, "Points-to analysis in almost linear time," in *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1996, pp. 32–41.

[15] M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, pp. 848–894, 1999.

[16] M. D. Weiser, "Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method," PhD Dissertation, The University of Michigan, Ann Arbor, 1979.

[17] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 171–180.

[18] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *WWW '04: Proceedings of the 13th international conference on World Wide Web*. New York, NY, USA: ACM, 2004, pp. 40–52.