1 Abstract

This project focuses on the finding the code portion prone to race conditions using the sequential proofs, that ensures that given assertions in the program holds in any sequential execution. After finding those code portions, we have used the proper locking scheme so as to ensure the logical concurrency of the program.

Given a sequential program annotated with assertions and proofs that ensures,assertion holds in any sequential execution of the program,we have used those assertions to synthesize proper locks in the program.Introduction of these locks,ensure that assertions that holds in sequential execution also holds in the presence of the concurrent execution,i.e this locking scheme ensure serializabilty.

For a concurrent library,to ensure consistency between different executions one way is to lock all the the statement of the program under single lock and other way could be to make each of statement atomic.The former one ensure isolation requirements but poor concurrency and with latter, library don't considers some of the safe interleavings between the threads.The paper *"Logical Concurrency Control from sequential proofs"* presents the idea that,what could be the safe interleaving of threads and how to find those safe interleaving using sequential proofs.

2 Paper Summary

2.1 Paper Idea

The paper presents the systematic process of finding unsafe interleaving of the threads starting form sequential proofs so as to meet the *"sequential specification"* of the library. These proofs ensures that given program satisfies certain assertions in the absence of concurrent client. And the assertions based on these proofs model can be used to synthesize the concurrent library. That further ensure these assertions are satisfied in the presence of the concurrent clients i.e enforcing the consistency constraint on the library.

In sequential setting the program can be viewed as Control Flow Automata $G = (V, E)$ where V set of program points and E set of edges where each edge represent a particular statement in the program.The proof μ for the sequential setting , can be viewed as invariant, i.e v ϵ V, satisfies μ(v); such that for each edge u$\rightarrow$v corresponding to the statement, say s,ensures that,given the precondition μ(u) holds,then after the execution of s μ(v) also holds. And if s is an assertion φ, then μ(u)$\Rightarrow \varphi$ should be valid. In a very broad way these constraints ensures that proof is valid for any sequence of execution of the sequential program. *"The invariant μ(u)at a vertex u indicates the property to hold at u to ensure that a sequential execution satisfies all assertions of library."*

Consider two concurrent threads say t^1 and t^2,the interleaving from thread,say t^1(at u) $\rightarrow$ t^2(at $\hat{u}$) is considered safe if execution of statement $\hat{s}$ in t^2 from $\hat{u} \rightarrow \hat{v}$ till the invariant μ(u) holds in t^1. But if the execution of $\hat{s}$ affects μ(u),interleaving needs to be restricted at u,as further execution of the s will fail to satisfy the assertion in t^1.

In more atomic sense the invariant μ can be thought of conjunction of boolean predicated say ρ_i and the check is made on predicates rather invariants. If the execution of the statement $\hat{s}$ of one thread invalidates the predicate ρ_j of the other thread of invariant μ(u),then locking u with the lock ℓ_{ρ_j} will allow only those threads to execute $\hat{s}$ that acquire lock ℓ_{ρ_j}. With this locking scheme we are ensuring isolation constraints over those statements which are prone to race condition basically unsafe interleaving.

2.2 Optimization

It may be possible that locking scheme here used can lead to redundant locks.e.g lock ℓ_1 is always held when lock ℓ_2 is acquired by any thread then lock ℓ_1 is redundant.Or it may be possible that particular statement don't invalidates any predicate in the library,then there is no need for having the lock for that predicate.Further the idea of reader-writer lock can also be introduced so as to increase the concurrency.

2.3 Example

Consider the following C code snippet.

```
1  int x,y;
2  int foo(int i)
3  {
4    x=x+i;                    // x=*
5    int j;                    // x=x+i
6    j=1;
7    x=x+1;                    // x=x+1
8    j=x+y;                    // j=x+y
9    return j;
10 }
```

If we consider the sequential execution of the above code,then we get the proofs in form of hoare triple e.g for the statement '4' $\{x = x^{in}\}x = x^{in} + i\{x = x^{in} + i\}$ forms a valid hoare triple. In concurrent library,these triples are used as the basis to find the proper locking scheme.Say t^1 and t^2 are two different threads at u is 4 and $\hat{u}$ is 7 respectively.

Consider the interleaving from $t^1 \rightarrow t^2$,then.Execution of t^2 invalidates the precondition $\{x = x^{in}\}$ at u. From t^1 perspective $\{x = x^{in} + 1\}x = x^{in} + i\{x = x^{in} + i\}$ don't forms a valid hoare triple,since it is not satisfiable for any value of x^{in}.

From the idea of the algorithm this is the implication that such of the interleaving must be controlled.If the statement u is held with a lock $\ell_{x=x^{in}}$ then t^2 will not be allowed to execute $\hat{u}$ until it acquire $\ell_{x=x^{in}}$.Since the thread t^1 holds the write lock at u other thread t^2 will not be allowed go through the statement $\hat{u}$. This ensure the atomicity of those statements which are dependent.

The Similar argument can be given for the statement 8 of t^1 with the statement 4 of thread t^2. Since the predicate $\{j = x + y\}$ is falsified.So the statement 8 and 4 are locked with $\ell_{j=x+y}$. So the Concurrent code with the locks looks like :

```
1  int x,y;
2  int foo(int i)
3  {
4    acquire(l1); acquire(l2);
5    x=x+i;
6    release(l1); release(l2);
7    int j;
8    j=1;
9    acquire(l1);
10   x=x+1;
11   release(l1); acquire(l2);
12   j=x+y;
13   release(l2);
14   return j;
15 }
```

The above output code can be further optimized,like the lock ℓ_2 is always acquired either after the lock ℓ_1 is acquired or it has been released.Giving the intuition that both the locks can be merged without affecting the correctness.

```
1  int foo(int i)
2  {
3    acquire(l12);
4    x=x+i;
5    release(l12);
6    int j;
7    j=1;
8    acquire(l12);
9    x=x+1;
10   j=x+y;
11   release(l12);
12   return j;
13 }
```

3 Implementation

To make the prototype implementation of the paper's algorithms we have used CPAChecker,Z3 SMT solver tools.The Process starts by getting the input i.e sequential proofs from CPAChecker. It is model checking tool that provide us with the set of prediactes for each program point,that are used as the basis as sequential proofs what we called as μ throughout. After finding those proofs. Algorithm presented in the paper then picks up a prediactes one after the other and checks whether or not the predicate is falsified,if yes then by which statement make that falsification. Now the statement is locked with that prediacte. Then our implementation uses these proofs and the sequential program to synthesize the concurrency control for the library.

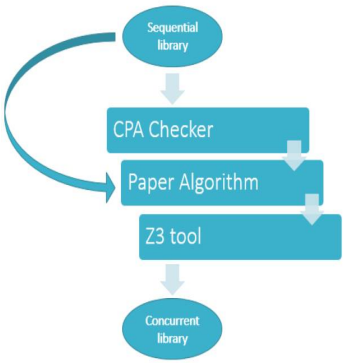

Figure1 is the High Level of the entire process model.

Sequential library is given as input to the cpachecker.The cpachecker is capable enough to obtain sequential proofs ,predicates on the basis of some of the assertions.These assertions are fed manually to the program. Since these proofs have dependency over these assertions,so we got to choose those assertions in the way that required property of the proof holds.

These proofs with the CFA are fed the algorithm presented in the paper. This prototype algorithm does the select a predicate from the proof's set and chcek it with all the statements in the program.Simply checking the validity of predicate after running the statement. This validity is checked using Hoare logic tool.If the statement even after execution satisfies the sequential specification then that statement is not locked. But if the specification are falsified then the statement is locked with the name of that predicate.

Z3 SMT solver is used to check whether $\{\mu(u) \wedge \varphi\}$ s ϵ E $\{\varphi\}$ is valid hoare triple or not. Z3 provides C++ interface that is harnessed to check for the validity of the hoare triple.Intuitively we want $\{\mu(u) \wedge \varphi\}$ and the statement 's' to get some post condition using the hoare logic for some,say assignment statement. And also

we know that $\{precondition\}statement\{postcondition\}$ forms a valid hoare triple,then it must imply our assertion/predicate statement φ. So if the post condition we obtained using the hoare logic of assignment statement implies our assertion then we are done.

The entire set of haore tripple that we need to check for,are obatined using using the protoype algorithm and saved into a particular file. And this file is given to Z3Smt solver to check for the valididty of the implication of prediacte.

The final set of locks with names of prediactes gives the required concurrent libarary.

4 Conclusion

The idea presented in the paper don't focuses on finding the lock based synchronization to ensure atomicity. Rather presented idea,focuses to avoid those interleaving that can make sequential specification invalid.So this locking scheme provides the lock in such a way that when the concurrent library is executed in the presence of these locks then all the sequential assertions holds at the end.

The idea is fairly simple and effective.And have been tested for variety of programs.The output concurrent library is efficient and correct for those tested programs.