All-du-path Coverage for Parallel Programs

- Cheer-Sun D. Yang Amie L. Souter Lori L. Pollock

Department of Computer and Information Sciences
University of Delaware, Newark

Presented by
Hyma S Murthy
Dated: 12/03/02

Main Idea..

- Automatic generation of All-du-paths for testing parallel programs.
- Introduce a tool "della pasta" (Delaware Parallel Software Testing Aid) for automatic generation of all-du-paths for shared memory parallel programs.

Introduction

- Parallel programs are categorized by their synchronization and communication mechanisms :
  - Message passing and shared memory
- Problems in testing parallel programs :
  - Non- deterministic nature prevents application of traditional testing approaches.
  - Lack of parallel software testing tools for testing correctness and reliability.

Contd…………

- Focus is on the applicability of all-du-path testing to parallel programs, and hence on generating test cases automatically for adequate testing.
- All-du-path (All-Definition Use-Path) coverage testing involves :
  - Identifying all du pairs in the program.
  - Create a path for each du pair.
  - Produce test data for testing the path.

Organization of the paper

- Program Model and Notation.
- Testing paradigm and dealing with nondeterministic nature of parallel programs.
- Problems in providing all-du-path coverage for shared memory parallel programs.
- Test Coverage classification.
- Du-Path finding algorithm.
- Della pasta tool.
- Conclusion.

Program Model

- Parallel program is considered to consist of multiple threads of control that can be executed simultaneously.
  - Thread is an independent sequence of execution.
- Communication between threads is through shared variables.
- Synchronization is achieved by calling ***post*** and ***wait*** system calls.
- ***Pthread_create*** system call is used for thread creation.

Notations

- Parallel program

  PROG = (T1,T2,…,Tn), where Ti, ($1 \le i \le n$) n(>2) represents threads. T1 is the manager and the rest are worker threads.
- Parallel Program Flow Graph – PPFG

  G = (V,E)

  V = nodes (statements in the program)

  E = (Es ? Et ? El)

  El = intra-thread control edges (m^i , n^i)

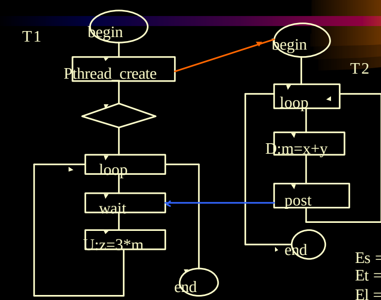
  Es = synchronization edges ($post^i$, $wait^j$)

  $post^i$ is post st in thread Ti and $wait^j$ is wait st in Tj (Ti ? Tj)

  Et = thread-creation edges (n^i, n^j)

  n^i is call st in Ti and n^j is the first st in Tj (Ti ? Tj)

Example of a PPFG



Es = arrow in blue
Et = arrow in red
El = arrows in yellow

Contd……..

- Path Pi (n_{u1}^i, n_{uk}^i) is an alternating sequence of nodes and intra-thread edges, e_{u1}^i, e_{u2}^i,…., e_{uk}^i.
- Du-pair is a triplet (var,n_u^i, n_v^j), n_u^i is the u^{th} node in thread Ti, where the *var* is defined, and n_v^j is the j^{th} node in thread Tj where it is used.
- A node nl ($1 \le l \le k$) in a parallel program is covered by a set of paths PATH = (P1,…Pk) in threads T1,T2,…Tk respectively or n_l ?$_p$ PATH, if n_l ?$_p$ P_l.
- MP(w) = {p | (p,w) ? Es}

  Matching posts for waits
- MP(p) = {w | (p,w) ? Es}

  Matching waits for posts

Last of the Notations !

- "a < b" – an instance of node a completes execution before an instance of node b.
- Du-path coverage for parallel programs
  - Given a shared memory parallel program PROG = (T1,T2,…,Tn), for each du-pair (var,n_u^i,n_v^j) in PROG, find a set of paths PATH = (P1,…Pk) in threads T1,T2,…Tk, that covers the du-pair (var,n_u^i, n_v^j), such that $n_u^i < n_v^j$.

Testing Paradigm

- Temporal testing is advocated for automatically generating and executing test cases in the face of nondeterminism.
- Alter the scheduled execution of program segments to detect synchronization errors.
- Temporal du-path testing involves identifying the delay points along the du-paths to be tested, and altering the execution time of process creation and synchronization events.
- Temporal Test case – TTC is a 3-tuple (PROG, I, D)

  PROG is program being tested, I is the input to it.

  D is the timing change, depending on which the execution time of synchronization events is changed for each test case.

Summary of the testing process

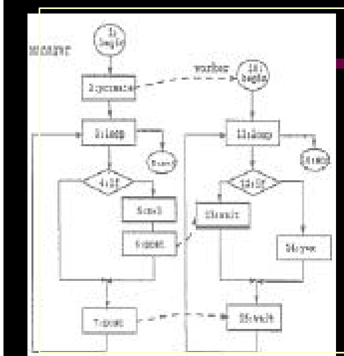
- Generate du-paths statically.
- Execute multiple times without timing changes.
- Examine trace results. Execution of different paths is an indication of synchronization errors.
- Generate temporal test cases for the du-paths and perform temporal testing.
- Examine the results.

Problems in all-du-path coverage

- Inconsistency in number of loop iterations may cause one thread to wait infinitely.

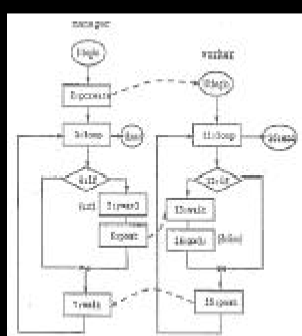
  Branch selection also influences thread termination.
- Define is after use
  - *Define* < *use* is violated
- This is not an exhaustive list however.

Du-pair coverage may cause an infinite wait



Path Coverage:
Manager: 1-2-3-4-5-6-7-3-8
Worker: 10-11-12-13-15-11-12-14-15-11-16

Du-pair is incorrectly covered



Path Coverage:
Manager: 1-2-3-4-5-6-7-3-8
Worker: 10-11-12-13-14-15-11-16

Test Coverage Classification

- Du-path coverage classified as
  - Acceptable and unacceptable
  - W-runnable and non-w-runnable
- Acceptability…. denoted as $PATH_a$

  A set of paths PATH for a du-pair (define, use) is acceptable if it satisfies the following:
  - define ?$_p$ PATH; use ?$_p$ PATH,
  - ? wait nodes w ?$_p$ PATH, ? a post node p ?$_p$ MP(w), such that p ?$_p$ PATH,
  - If ? (post, wait) ? Es, such that define < post < wait < use, then post, wait ?$_p$ PATH.
  - ? n^j ?$_p$ PATH where (n^i, n^j) ? Et, ? n^i ?$_p$ PATH.

W-runnability of du-path coverage…$PATH_w$

- W-runnable path coverage doesn't cause infinite wait in any thread. $PATH_a$ is w-runnable if following conditions are satisfied :–
  - Each instance of a wait, w^t_i ?$_p$ PATH, ? an instance of post, p^s_u ?$_p$ PATH, where p^s_u ? MP(w_i^t).
  - ?/ post nodes p^j, p^i, and wait nodes, w^i, w^j such that

    ((p^i < w^j) ? (p^j < w^i)) ? (w^i < p^j) ? (w^j < p^i)

A peek at related work

- The du-path finding algorithm for parallel programs is a combination of the Depth first search (DFS) approach and the Dominator (DT) and Post-dominator (also Implied tree -IT) trees approach.
- The DFS and the DT-IT approaches are designed for sequential programs. DFS finds a path to connect two nodes, and DT-IT approach finds branch coverage.
- Individually, when applied to parallel programs, they fail to provide coverage for intervening wait's and their matching posts as required for $PATH_a$ or may generate a path where define is after use.

The Hybrid Approach

- Uses two sets of disjoint nodes :
  - *Required nodes* which include the pthread_create() call nodes, the define node, the use node to be covered, and the associated post and wait nodes such that the partial order define < use is guaranteed.
  - *Optional Nodes*, which are the remaining nodes along the path, whose order is not set.
- The algorithm has two phases :
  - *Annotate phase*, where DFS is used to cover required nodes, DT-IT is used to cover optional nodes. Once a path to a node is found, all nodes along the path are given a number, TRN, traversal control number.
  - *Path Generation phase*, where the actual path is generated using the TRNs.

Annotate_the graph()

Input: A DU-pair, and a PPFG
Output: Annotated PPFG
Mcthod:

1. Initialize TRN's, decision queues, and working queues;
2. Find a path to cover pthread-create and define nodes using dfs:
   From the define node, search for the use node using dfs;
3. Complete the two sub-paths using DT-IT.
4. For each node in the complete paths:
   Increment TRN by one:
   If node is a WAIT,
      Add matching nodes into appropriate working queues,
   If node is an if-node,
      Add the successor node in the path into decision queue;

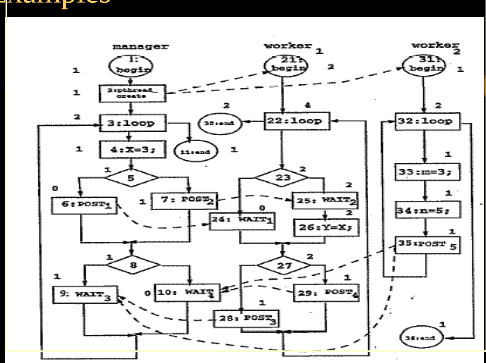
```
5. /* process the synchronization nodes * /
while (any working queuePot empty)
{       For each thread, if working queue not empty
        {       Remove one node from the working queue;
                if the node's TRN is zero
                {       Find a path to cover this node
                        For each node in the complete path:
                        Increment TRN by one;
                If node is a WAIT,
                Add matching nodes into appropriate working queues,
                If node is an if-node,
                Add the successor node in the path into decision queue;
                }
        }
}
```

Traverse_the_graph()

Input: An annotated PPFG
Output: A DU-path
Method:

```
For all threads
{   current = begin node of the thread:
    while (current node's TRN > 0 and current is not the end node)
    {   add the current node to the result DU-path;
        decrement TRN of current node by one:
        if ( current is an if-node )
                current = first node from decision queue;
                delete the first node in the queue:
        else if ( current is n loop node )
                current = successor with smallest non-zero RPO:
        else
        current = successor node of current;
    }
}
```

Examples



Generating $PATH_a$

- The definition of X at node 4 and its use at node 26 is considered.
- Create a path between pthread_create(), define, post2,wait2, and the use nodes using DFS.
- Using the DT -IT approach, get the paths 1-2-3-4-5-7-8-9-3-11 for the manager and 21-22-23-25-26-27-28-22-30 for the worker1.
- TRNs are assigned to all the nodes above, 22 has TRN 2 and all others have TRN 1.
- When node 9 is reached nodes 28 and 35 are placed into the working queues for worker1 and worker2.
- Node 28 is covered, so a path 31-32-33-34-35-32-36 is found for the worker2.
- Phase 2 finds the final paths for all the threads, which are the above.

Generating non-$PATH_w$

- The steps till generating the path for the manager thread remain the same as the previous example.
- The path for the worker1 thread is generated as 21-22-23-25-26-27-29-22-30. Node 29 is covered instead of 28.
- When node 9 is reached during the traversal, a path is generated for both nodes 28 and 35 as
  - 21-22-23-25-26-27-28-22-30 and
    31-32-33-34-35-32-36 respectively.
- Hence the second phase generates the following paths :
  Manager : 1-2-3-4-5-7-8-9-3-11
  Worker1 : 21-22-23-25-26-27-29-22-23-25-26-27-28-22-30
  Worker2 : 31-32-33-34-35-32-36
- Worker1 has an infinite wait, hence not w-runnable.

Correctness of the Algorithm

- TRN preserves the no of traversals of a node within a loop.
- TRN and the decision queue, guarantee that the same sequence of branches traversed during the first phase will be selected during the second phase.
- DFS ensures that define < post < wait < use.
- TRN and working queues guarantee the termination of the algorithm – this is proved by means of induction on the pairs of synchronization nodes.
- Using the above, given a du-pair, the hybrid approach terminates and finds a $PATH_a$.

The Tool - “Della Pasta”

- Objective is to demonstrate partial automation of test data generation and respond to programmer queries on testing.
- Functions include finding all du-pairs, finding path coverage for user specified du-pairs, displaying all-du-path coverage in graphic mode or text mode and adjusting path-coverage when desired by the user.
- Uses a static analyzer to perform the first two functions, and a path handler for the other two.

Conclusions

- *Limitations*
  - The algorithm requires that PPFG be constructed statically, else the analysis may not produce meaningful du-pairs.
  - In case of a clear before/after wait, the algorithm reports more test cases than needed.
- *Successes*
  - First attempt at extending sequential testing criteria to parallel programs.
  - Classifies coverage, identifies problems in the parallel program realm and finds all-du-path coverage for shared memory parallel programs.