Full Length Research Paper

Generic method for statistical testing of parallel programs based on task trees

Miroslav Popovic1*, Ilija Kupresanin2 and Ilija Basicevic1

1University of Novi Sad, Fruskogorska 11, 21000 Novi Sad, Serbia.
2JP SRBIJAGAS, Narodnog Fronta 12, 21000 Novi Sad, Serbia.

Accepted 13 March, 2012

This paper deals with a particular class of parallel programs, which are based on task trees. The main objective of this paper was to adapt the generic method for statistical testing of sequential programs (GMST-SP) for this class of parallel programs, such that adapted method (GMST) can treat a family of task trees rather than just a single task tree, and that it can respect various evolutions of individual task trees. In this paper, we compare GMST with the adapted exhaustive testing method (ET) and with the previously adapted statistical usage testing method (SUT), based on experimentally measured testing effort and path coverage. GMST and SUT both have better deep path coverage than ET. SUT requires less testing effort than GMST and ET, but its deep path coverage decreases with the number of tasks. Finally, GMST has advantage over SUT because it provides constant mean level of deep path coverage, which can be regulated by the required testing quality.

Key words: Multicores, parallel programming, parallel programs, structural testing, statistical testing.

INTRODUCTION

In this paper, we deal with a class of parallel programs based on task trees, which is referred to as the Task Tree Executor (TTE) task trees, (Popovic, 2009). A TTE task tree is a tree of tasks such that child nodes of the same parent are mutually independent. An example of the real-world application based on task trees is the large-scale electric power distribution management system (Basicevic, 2009). Conceptually, TTE task trees are very much similar to Intel Cilk plus directed acyclic graphs of Cilk strands (Intel, 2011a) and Intel Thread Building Blocks task graphs (Intel, 2011b), which are the foundations of the latest Intel Parallel Studio 2011 commonly used within Microsoft visual studio.

Testing and verification of parallel programs based on task trees is a new challenge for the research community. Existing and rather well established methods and accompanying tools for testing sequential programs, as well as for multithreaded and message-passing programs (Souza, 2011) are the good starting point, but they are not directly applicable to task trees. Obviously, methods for sequential programs might be applicable only on the level of a single TTE task. On the other hand, existing multithreaded methods are targeting programs with coarse-grained interdependent threads based on pthreads API, whereas TTE task trees consist of fine-grained independent tasks that use TTE API. Alternatively, methods for message-passing programs are targeting single program multiple data (SPMD) programming model wherein processes executed on a geographically distributed computers execute different partitions of loop iteration space, whereas TTE task trees are typically performing parallel processing based on geographic decomposition of large arrays. Therefore, there is a need to adapt the existing methods for this new kind of parallel programs, and maybe explore new approaches and invent new methods. Recently, Wolf’s approach to statistical usage testing (SUT) and reliability estimation (Wolf, 1993a, b, 1994a, b; Broakman, 2002)
has been adapted for testing large-scale task trees based on operational profiles (Popovic, 2010). In this paper, we present two novel methods for testing task trees, which we developed by adapting the traditional methods for testing sequential programs.

Firstly, we adapted exhaustive testing of sequential programs (ET-SP), which is the most prominent deterministic, a.k.a. controlled, testing method. The adapted method known as ET is essentially used as the baseline for comparison analysis. A useful corollary of ET is that we may reuse the mechanisms of controlled execution of task trees for any other kind of controlled testing. Secondly, we took the rather well established generic method for statistical testing of sequential programs (GMST-SP), which promotes an interesting idea of using random generation of combinatorial structures, such as graphs, trees, words, paths, etc., for statistical testing of sequential programs (Denise, 2004; Gouraud, 2005) and adapted it into the GMST of task trees. Thirdly, we compare ET and GMST with SUT. Note that all the methods are being applied on the same level of abstraction, that is, the level of TTE task trees.

These three testing methods are evaluated in the paper by applying them to test all the TTE task tree families that may be constructed by the given number of tasks. Evaluation is based on the testing effort and the path coverage statistics. GMST and SUT both have better deep path coverage than ET. SUT requires less testing effort than GMST and ET, but its deep path coverage decreases with the number of tasks. Finally, GMST has advantage over SUT because it provides constant mean level of deep path coverage, which does not depend on the number of tasks, and which can be regulated by the required testing quality $q$, but at the expense of greater testing effort that might be tolerable for mission-critical projects.

This paper, however, aims only to compare the three methods and since that can be successfully done by experiments on smaller task trees, we restricted our experiments to tasks trees consisting of seven tasks. The advantage of this approach is that we could use simple PCs available in our laboratory. It seems appropriate mentioning that experiments with bigger task trees would require application of powerful servers, but such experiments would not make a difference in our conclusions.

**Related work**

Some approaches to testing parallel programs try to reduce it to testing sequential equivalents. For example, Sung (1988) used preprocessor translator and path finder to reduce parallel program testing to testing of the translated serial program. Alternatively, Bocchino et al. (2009) argued for a parallel programming model that is deterministic by default: deterministic behavior is guaranteed unless the programmer explicitly uses nondeterministic constructs, and they claim then that the same test suites developed for the sequential code can be used for such parallel code. In contrast, in this paper, we directly test parallel programs written in the state-of-the-art programming languages.

Structural testing or white-box testing, of sequential programs is a well established testing technique that derives test cases from the logical structure of the program under test (Rapps, 1985). The test cases are derived in such a way that certain elements in the program structure must be covered during testing. Typical structural testing criteria are all statements (all-nodes), all branches (all-edges) and all variable definition-use pairs in the program. It is worth mentioning that structural testing is declared mandatory in the existing regulations for the safety-critical systems (Zhu, 1997).

More recently, Augustio et al. (2007) have proposed a family of control flow and data flow based testing criteria for aspect-oriented programs. Alternatively, Jee et al. (2009) have found that classical coverage criteria, based on control flow graphs are inadequate when applied to a data flow language, such as the functional block diagram (FBD). As a solution, they proposed the basic coverage, input condition coverage, and complex condition coverage. Both papers (Augustio, 2007; Jee, 2009) presented interesting extensions of the traditional structural testing of sequential programs.

Yang and Chung (1990) proposed a model to represent the execution behavior involving concurrent paths (C-paths) and concurrent routes (C-routes) of a concurrent program, and described a test execution strategy to detect various faults in a concurrent program, but the actual methodologies for the selection of C-paths and C-routes are not presented in the paper. Further on, Yang and Pollock (2003) presented a testing framework for all uses testing coverage of parallel programs, which is applicable to conventional multithreading programs, but is not applicable to TTE based programs. Alternatively, in this paper, we use the given task tree as the control flow model and all the task tree evolution paths (all-paths) as the coverage criteria. Since this testing method covers all the paths, we refer to it as ET method.

The GMST-SP is the second testing method used in this paper. GMST-SP was initially inspired by the work of Thevenod-Fosse and Waeselynck (1991), wherein the choice of the distribution on the input domain was guided by some coverage criteria of either the program (structural statistical testing) or some specification (functional statistical testing). Another source of inspiration for GMST-SP was the work of Flajolet et al. (1994) who generalized and systematized efficient algorithms for generating uniformly, and randomly a variety of combinatorial structures, originally developed by Wilf (1977) and Nijenhuis (1979).

GMST-SP (Denise, 2004) is the method for statistical
testing according to a given description of the behavior of the system under test. It uses uniform random generation routines for drawing paths from the set of system execution paths. Next, a constraint resolution step is performed in order to construct a set of test data that activate the generated paths. GMST-SP also uses linear programming technique to improve the probabilities for the elements to be covered by testing. The inventors of GMST-SP demonstrated applicability of their method and accompanying tools in a research where they conducted more than 10,000 experiments on industrial software (Gouraud, 2005).

The SUT is the third testing method used in this paper. Woit (1993a, b, 1994a, b) did much of the work in this area in her Ph.D. thesis and related papers. It is worth mentioning that Woit’s approach to automated test case generation and software reliability estimation based on operational profiles has been recognized as a de facto standard and a paradigm accepted by industry (Brookman, 2002).

Woit’s work has been followed in the area of model-based testing using the generic modeling environment (Popovic, 2005, 2006a, b, 2007). More recently, Woit’s approach has been adapted for testing large-scale task trees (Popovic, 2010). Of course, many other researchers have been active in the area of SUT. An example is a rather promising fuzzy logic based approach presented by Kumar et al. (2007). Generally, model based testing is being an active playground for many researchers.

It is worth mentioning that we did not use the adaptive random testing method (ART) (Chen, 2004a, b, 2007) in this paper because it is known that it involves a significant additional effort in generating required test cases. The inventors of ART state in their initial work (Chen, 2004b) that ART outperformed ordinary random testing (ORT) by a factor from 1 to 50% on a benchmark of 12 error-seeded programs. But, the cost to achieve this result was rather high – effectively 10 times more test cases would have to be generated than it would be generated by ORT.

**TTE task trees and their use**

A TTE task tree is a program structure constructed and executed by making use of the TTE API (Popovic et al., 2009), which consists of the following C functions:

1) int TS_CreateTaskGraph(long troot, FNptr bu, FNptr td, long nthreads);
2) int TS_AddTask(long parent, long task);
3) int TS_DeleteTask(long task);
4) int TS_ExecuteBottomUp(void);
5) int TS_ExecuteTopDown(void);
6) void TS_DestroyTaskGraph(void);  

For example, Program 1 creates and executes a simple task tree. Program 1 first creates the initial task tree with the root task 0, and specifies the bottom-up callback function *CalcCurrents*, the top-down callback function *CalcVoltages*, and sets the number of threads to 8 (line 2). Subsequently, it adds root’s children, namely tasks 1, 2 and 3 (lines 3 to 5); adds children of task 2, these are tasks 4 and 5 (lines 6 to 7); executes the task tree bottom-up in parallel (line 8); executes the task tree top-down in parallel (line 9); and, finally, destroys the task tree (line 10). When a task is executed, TTE calls an appropriate callback function and passes it the corresponding task’s identification. For example, if task 2 is to be executed bottom-up, TTE calls the callback function *CalcCurrents* and passes it the identification (ID) 2; similarly, if task 3 is to be executed top-down, TTE calls the callback function *CalcVoltages* and passes it the ID 3.

**Program 1**

```c
1) void main(void) {
2)   TS_CreateTaskGraph(0, CalcCurrents, CalcVoltages, 8);
3)   TS_AddTask(0, 1);
4)   TS_AddTask(0, 2);
5)   TS_AddTask(0, 3);
6)   TS_AddTask(2, 4);
7)   TS_AddTask(2, 5);
8)   TS_ExecuteBottomUp();
9)   TS_ExecuteTopDown();
10)  TS_DestroyTaskGraph();
11) }
```

Originally, TTE was introduced to enable parallelization of legacy FORTRAN code for calculations on electricity power distribution networks. These networks are typically modeled as network trees, wherein the tree root models a transformer that is a part of the high-voltage transmission system, intermediate tree nodes model mid and low voltage transformers, tree edges model transmission lines, and tree leaves model end power consumers (industry, homes, etc.). The sequential load flow calculation is typically performed in two subsequent network tree traversals. The first traversal is bottom-up and it uses the first Kirchhoff law to calculate all the currents in the network. The second traversal is top-down and it uses the second Kirchhoff law to calculate all the voltages in the network. The most of the other calculations on electricity power distribution networks are performed in the same manner.

TTE parallelization concept is based on slicing the network tree into slices of a given depth and creating the corresponding tasks that are responsible to do the calculations on individual network tree slices. The network tree slice depth is defined by the number of intermediate tree edges, starting from the top of the slice.
and ending at the bottom of the slice. The result of slicing is a task tree that corresponds to network tree. The advantage of this approach is that certain tasks within the task tree may be executed in parallel, which leads to parallel processing of the corresponding data stored in the network tree model.

Although TTE task trees may look like a very specialized kind of parallel programs, actually the same kind of program structures are found in many similar applications, for example, gas and oil distribution networks, water distribution networks, etc. Therefore, the methods presented in this paper should be directly applicable to this broad class of critical infrastructures.

MATERIALS AND METHODS

ET method

Definition 1

A task \( r \) is a callback function that executes as a local OS thread.

Definition 2

A task tree is an undirected radial (that is, acyclic) graph of tasks \( TG \) whose nodes are tasks interconnected with links which indicate predecessor-successor relations. A task tree consists of a set of \( k \) tasks \( TK = \{ t_1, t_2, \ldots, t_k \} \), and a set of \( (k-1) \) links \( L = \{ l_1, l_2, \ldots, l_{k-1} \} \).

Definition 3

A root \( rt \) is a predecessor of all the nodes in a task tree.

Definition 4

A task tree leaf is a task tree node that has no successors.

Definition 5

For any two directly connected nodes in a task tree, the node closer to the root is a predecessor of the other node, and the other node is its successor.

A parallel task tree execution may be formally described as a composition of two operators, namely \( P() \) and \( S() \), where the former represent parallel execution of its parameters, whereas the latter corresponds to sequential execution (Basiciv, 2009). As already mentioned, a task tree can be executed in parallel top-down or bottom-up.

Definition 6

A task tree execution path or a path in a task tree or a trace, is a sequence of terminations of individual tasks \( t_1, t_2, \ldots, t_k \) during the task tree execution. The length of this sequence for the task tree composed of \( k \) tasks is always equal to \( k \).

Definition 7

A test case is a single task tree execution described by the corresponding path.

Definition 8

A task tree evolution graph consists of all the paths of a given task tree with a common root node.

Traversing an evolution graph top-down, that is, following a path from the root towards some leaf, corresponds to one particular top-down evolution of the given task tree, whereas traversing an evolution graph bottom-up, that is, in the opposite direction, corresponds to one particular bottom-up evolution of the given task tree. Since top-down and bottom-up evolutions of the given task tree are completely symmetrical, we will consider only the former, without any losing generality.

Definition 9

A task tree forest is a series of task trees of the same complexity (that is, comprising the same number of nodes) that is generated as a test suite.

Definition 10

A task tree family is a task tree forest containing all the task trees of the same complexity.

When conducted for a given number of tasks, the ET method comprises the following steps:

1) Generate the task family tree (using Algorithms 1 to 3).
2) For each task tree within the generated task family do step a through d.
   a) Create the evolution graph (using Algorithm 4).
   b) Collect all the paths (using Algorithms 5 to 7).
   c) Convert all the paths into the corresponding task schedules.
   d) Execute all the schedules on the target OS simulator.

In the text that follows, we describe the details of Algorithms 1 to 7. The intuition for Algorithms 1 to 3 comes from observing the task tree family genealogy (Figure 1). Figure 1 shows the four task tree families that are consisting of one, two, three and four tasks, respectively. The family comprising one task has just one member, the task tree \( T_1 \). The family comprising two tasks has also one member, the task tree \( T_1-2 \). The family comprising three tasks has two members, which are the task trees \( T_1-2-1 \) and \( T_1-2-2 \). Finally, the family comprising four tasks has six members, which are the task trees \( T_1-2-1-3-1 \), \( T_1-2-1-3-2 \), \( T_1-2-1-3-3 \), \( T_1-2-2-3-1 \), \( T_1-2-2-3-2 \) and \( T_1-2-2-3-3 \). Labeling of the task trees was done with the intention to show the links between the nodes, which is the key to understand how all the task trees are constructed.

Start by considering the first task tree family. It has just one task, so we can construct it very easily, just by putting that task in the tree, which is then immediately finalized, and we have also immediately the complete task tree family comprising one task. Once we have the first family we easily construct the second family. How? Simply by taking the second task and connecting it with the first one. By doing so we get the second family, which like the first family has only one member. But, by doing so, we also get the idea how to construct the next family from the previous one. Simply take the new task and create the new tree by connecting the new task to one of the tasks in the previous tree. By repeating this step, systematically, for all the tasks in the previous tree, we get all the trees in the new family, that is, we effectively construct the new family. Let’s check this approach for the third family.

We take the second family, which has just the tree \( T_1-2 \). Then we
Figure 1. The task tree family genealogy.

take the task 3 and connect it to the task 1, and we effectively construct the tree T1-2.1-3. Next, we take the task 3 and connect it to the task 2 to construct the tree T1-2.2-3, and we are done. The third family is successfully constructed. Similarly, by taking the task 4 and keep connecting it to each of the task 1, 2 and 3, once in turn, and for both of the trees in the third family, we construct the
The function \textit{GenerateTreeFamily} in Algorithm 1 starts from the empty task tree family (lines 2 and 3). Each subsequent recursive call to the function \textit{GenerateTreeFamilyR} (line 15) generates the next task tree family by introducing the next task (starting from the task number 1), until the desired family is constructed (line 16). The next family is constructed from the previous family in a rather simple way. The function \textit{GenerateTreeFamilyR} in Algorithm 1 takes task trees from the previous family one by one (line 11) and calls the function \textit{AddTaskToEachNode} in Algorithm 2 (line 12), which in turn creates new task trees by introducing the next task and connecting it to the task already connected in the previous task tree (lines 19 to 22). The functions \textit{GenerateTreeFamilyR} and \textit{AddTaskToEachNode} repeat this routine for all the task trees in the previous family and for all the tasks already connected in the task trees.

\textbf{Algorithm 1}

1) \textit{GenerateTreeFamily}(\textit{noTasks}) =
2) \hspace{1em} \textit{TreeFamily empty}
3) \hspace{1em} \textit{GenerateTreeFamilyR}(\textit{noTasks}, \textit{1}, \textit{empty})
4) \hspace{1em} \text{for} \textit{curTask} in \textit{noTasks}
5) \hspace{2em} \textit{GenerateTreeFamilyR}(\textit{noTasks}, \textit{curTask}, \textit{TreeFamily curFamily}) =
6) \hspace{3em} \textit{TreeFamily fam}
7) \hspace{4em} \text{if} \textit{curTask} = 1
8) \hspace{5em} \textit{ts} = \textit{new Task(1)}
9) \hspace{5em} \textit{fam} = \textit{fam} \cup \textit{ts}
10) \hspace{3em} \text{else}
11) \hspace{4em} \text{for} \textit{each} \textit{task tree} \textit{tr} in \textit{task family} \textit{fam}
12) \hspace{5em} \textit{fam} = \textit{AddTaskToEachNode}(\textit{curTask}, \textit{tr}, \textit{fam})
13) \hspace{4em} \text{DestroyTreeFamily}(\textit{curFamily})
14) \hspace{3em} \text{if} \textit{curTask} < \textit{noTasks}
15) \hspace{4em} \textit{fam} = \textit{GenerateTreeFamilyR}(\textit{noTasks}, \textit{curTask}+1, \textit{fam})
16) \hspace{3em} \textit{fam}

The function \textit{AddTaskToEachNode} in Algorithm 2 uses the function \textit{CloneTree} in Algorithm 3 to clone a given task tree. It also uses the task tree method \textit{locateTask(id)} to locate the task with the identification \textit{id} (line 20), the method \textit{addNode(n)} to connect the task to the new task \textit{n} (line 21), and the constructor \textit{Task(taskid)} to construct the new task with the identification \textit{tasked} (line 21).

\textbf{Algorithm 2}

17) \textit{AddTaskToEachNode}(\textit{taskid}, \textit{Tree tree}, \textit{TreeFamily family}) =
18) \hspace{1em} \text{for} \textit{id} \in \{1, \text{taskid}\}
19) \hspace{2em} \textit{clone} = \text{CloneTree}\textit{(tree)}
20) \hspace{2em} \textit{tk} = \textit{clone}\textit{}.\textit{locateTask(id)}
21) \hspace{2em} \textit{tk}.\textit{addNode(new Task(taskid))}
22) \hspace{1em} \textit{family} = \textit{family} \cup \textit{clone}
23) \hspace{1em} \textit{family}

The function \textit{CloneTree} in Algorithm 3 effectively clones the given task tree. It does this by creating the root task (line 25) and adding its successors, and successor's successors, through recursive calls to the function \textit{CloneSubTree} (line 35). It uses the task tree method \textit{getSuccessors} to get all the successor tasks of a given task (lines 29 and 33).

\textbf{Algorithm 3}

24) \textit{CloneTree}(\textit{Tree tree}) =
25) \hspace{1em} \textit{clone} = \textit{new Task(1)}
26) \hspace{1em} \textit{clone} = \text{CloneSubTree}(\textit{tree}, \textit{clone})
27) \hspace{1em} \textit{CloneSubTree}(\textit{Task task}, \textit{Task clone}) =
28) \hspace{2em} \textit{ttsucc} \leftarrow \textit{task}.\textit{getSuccessors()}
29) \hspace{2em} \text{for} \textit{each} \textit{task} \textit{ti} in \textit{ttsucc}
30) \hspace{3em} \textit{clone}.\textit{addNode(new Task(ti.getId()))}
31) \hspace{2em} \textit{ttsucc} \leftarrow \textit{clone}.\textit{getSuccessors()}
32) \hspace{2em} \text{for} \textit{each} \textit{ti} in \textit{ttsucc} \& \text{each corresponding} \textit{ci} in \textit{ctsucc}
33) \hspace{3em} \text{CloneSubTree}(\textit{ti}, \textit{ci})

The function \textit{TDET\textunderscore CreateEvolutionTree} in Algorithm 4 works to generate the evolution graph that is the non existing (virtual) task with both the internal identification \textit{id} and the external \textit{eid} equal to 0 (lines 39-40). The variable \textit{enode} contains the first still not assigned external evolution graph node identification (line 41). The external identification is assigned by the method \textit{setEid} (lines 40 and 50). The function \textit{TDET\textunderscore CreateEvolutionTree} then starts a series of (self) recursive calls to the function \textit{CreateEvolutionTreeR} in Algorithm 4, which in turn clones individual tasks from the original task tree (lines 49 to 50), and connects these clones according to all the possible top-down evolutions of the given task tree (lines 55 to 60).

\textbf{Algorithm 4}

36) \textit{TDET\textunderscore CreateEvolutionTree}(\textit{Task task}) =
37) \hspace{1em} \textit{tasklist} \leftarrow \{\}
38) \hspace{1em} \textit{enode} \leftarrow 0
39) \hspace{1em} \textit{tdegraph} \leftarrow \textit{new} \textit{Task}(0)
40) \hspace{1em} \textit{tdegraph}.\textit{setEid}(\textit{enode})
41) \hspace{1em} \textit{enode} \leftarrow \textit{enode} + 1
42) \hspace{1em} \textit{CreateTDEvolutionTreeR}(\textit{tdegraph}, \textit{task}, \textit{enode})
43) \hspace{1em} \textit{tdegraph}.\textit{deleteNode}(0)
44) \hspace{1em} \textit{tdegraph}
45) \hspace{1em} \textit{tdegraph}
46) \hspace{1em} \textit{CreateEvolutionTreeR}(\textit{Task parent}, \textit{Task task}, \textit{noPaths})
47) \hspace{2em} \textit{tasklist} \leftarrow \{\}
48) \hspace{2em} \textit{aliveSuccessors} \leftarrow \textit{aliveSuccessors} \cup \{\textit{np}\}
49) \hspace{2em} \textit{clone} = \textit{new} \textit{Task( task.getId() )}
50) \hspace{2em} \textit{clone}.\textit{setEid}(\textit{enode})
51) \hspace{2em} \textit{enode} \leftarrow \textit{enode} + 1
52) \hspace{2em} \textit{parent}.\textit{addChild}(\textit{clone})
53) \hspace{2em} \textit{aliveSuccessors}.\textit{remove}(\textit{clone})
54) \hspace{2em} \textit{ts} \leftarrow \textit{task}.\textit{getSuccessors()}
55) \hspace{2em} \text{for} \textit{each} \textit{np} in \textit{ts}
56) \hspace{3em} \textit{aliveSuccessors} \leftarrow \textit{aliveSuccessors} \cup \{\textit{np}\}
57) \hspace{2em} \text{for} \textit{each} \textit{np} in \textit{aliveSuccessors}
58) \hspace{3em} \textit{parent} \leftarrow \textit{CreateEvolutionTreeR}(\textit{clone}, \textit{np}, \textit{aliveSuccessors})
59) \hspace{2em} \textit{parent}

The function \textit{TDET\textunderscore GetPaths} in Algorithm 5 uses the function \textit{updateNoPaths} in Algorithm 6 to update the number of paths crossing the node for each node of the given evolution graph (line 65), as well as the function \textit{getNextPath} in Algorithm 7, which returns the next path (line 67), starting from the first one and ending with the last one.

\textbf{Algorithm 5}

62) \textit{TDET\textunderscore GetPaths}(\textit{Task tdegraph}) =
63) \hspace{1em} \textit{paths} \leftarrow \{\}
64) \hspace{1em} \textit{path pt} \leftarrow \textit{null}
65) \hspace{1em} \textit{nopaths} \leftarrow \textit{updateNoPaths}(\textit{tdegraph})
66) \hspace{1em} \text{for} \textit{i} \in \{0, \text{nopaths}\}
condition for uniformity, which requires \( \log \frac{1}{\left| \mathcal{P} \right|} \) to be large enough. The function \( \text{getNextPath} \) in Algorithm 6 is a self recursive function, which uses the variable \( \text{acc} \) to count the number of paths crossing the given node of the evolution graph (lines 74 and 75) and the method \( \text{setNoPaths} \) to set that value in the corresponding node’s attribute (line 77).

**Algorithm 6**

\[
\begin{align*}
70) & \quad \text{updateNoPaths}(\text{Task} \ root) = \\
71) & \quad \text{acc} \leftarrow 0 \\
72) & \quad s \leftarrow \text{root}.\text{getSuccessors}() \\
73) & \quad \text{if } s = \{ \} \\
74) & \quad \text{acc} \leftarrow 1 \\
75) & \quad \text{else for each } t \in s \\
76) & \quad \text{acc} \leftarrow \text{acc} + \text{updateNoPaths}(t) \\
77) & \quad \text{root}.\text{setNoPaths}(\text{acc}) \\
78) & \quad \text{acc}
\end{align*}
\]

The function \( \text{getNextPath} \) in Algorithm 7 constructs the path through a series of self recursive calls (line 89). It uses the methods \( \text{getPaths} \) and \( \text{setNoPaths} \) to decrement the remaining number of paths crossing the given evolution graph node (line 86).

**Algorithm 7**

\[
\begin{align*}
79) & \quad \text{getNextPath}(\text{Task} \ r, \ \text{Path} \ ip) = \\
80) & \quad \text{path} \leftarrow \text{ip} \\
81) & \quad \text{path}.\text{append}(\text{r}.\text{getId}()) \\
82) & \quad s \leftarrow \text{r}.\text{getSuccessors}() \\
83) & \quad \text{if } s = \{ \} \text{ return path} \\
84) & \quad \text{acc} \leftarrow 1 \\
85) & \quad \text{for each } t \in s \\
86) & \quad \text{acc} \leftarrow \text{acc} + \text{updateNoPaths}(t) \\
87) & \quad \text{path} = \text{getNextPath}(t, \ \text{path}) \\
88) & \quad \text{break} \\
89) & \quad \text{path}
\end{align*}
\]

**Original GMST-SP method**

**Definition 11**

Let \( v_s \) be the starting vertex of a control flow graph (CFG) and \( v_e \) be the ending vertex of the CFG, such that for any vertex \( v \), there is a path from \( v_s \) to \( v \) and a path from \( v \) to \( v_e \) in CFG.

**Definition 12**

If \( n \) is a positive integer, \( P_n \) (resp. \( P_{\leq n} \)) denotes the set of paths of length \( n \) (resp. of length less or equal to \( n \)) in CFG from \( v_s \) to \( v_e \), and \( P_{\geq n} \) denotes the whole set of paths from \( v_s \) to \( v_e \).

**Definition 13**

Given any vertex \( v \), let \( f_v(m) \) denote the number of paths of length \( m \) which connect the vertex \( v \) to the end vertex \( v_e \).

Assume that at some step of generating a path, we are on vertex \( v \), which has \( k \) successors \( v_1, v_2, ..., v_k \). Also assume that \( m > 0 \) edges remain to be crossed in order to get a path of length \( n \) to \( v_e \).

Then the probability of selecting a vertex \( v_i \) is equal to \( f_{v_i}(m - 1) / f_{v}(m) \) in order to satisfy the condition for uniformity, which requires that the probability of going to any successor of \( v \) must be proportional to the number of paths of suitable length from this successor to \( v_e \). The numbers \( f_{v}(i) \), for every \( 0 \leq i \leq n \) and every vertex \( v \) of the CFG, are computed in practice using the following recurrence rules:

\[
f_{v}(0) = 1, \quad f_{v}(i) = \sum_{v' \in v} f_{v'}(i - 1) \text{ for } i > 0
\]

where \( v \rightarrow v' \) means that there is an edge from \( v \) to \( v' \). Note that \( v \) is equal to \( v' \) in case of a loop in the CFG.

In the next definition we will use the following notation:
1) \( \mathcal{D} \) is a description of a system under test, which is based on a combinatorial structure, for example, a CFG.
2) \( \mathcal{C} \) is a given coverage criteria, such as all-vertices, all-edges, all-paths, etc. (Note that in this paper we are interested in the all-paths criterion).
3) \( \mathcal{E}(\mathcal{D}) \) is a set of elements of the graph \( \mathcal{D} \), which must be traversed at least once, by the corresponding set of test cases, in order to satisfy the given criteria \( \mathcal{C} \).

**Definition 14**

Let \( q_{\mathcal{C} | \mathcal{M}}(\mathcal{D}) \) be the quality of the method with respect to \( \mathcal{C} \), which is defined as the minimal probability of covering any element in \( \mathcal{E}(\mathcal{D}) \) when drawing \( N \) test cases:

\[
q_{\mathcal{C} | \mathcal{M}}(\mathcal{D}) = 1 - (1 - \frac{1}{|\mathcal{P}_{\leq n}|})^N
\]

where \( q_{\mathcal{C} | \mathcal{M}}(\mathcal{D}) \) is the quality for a single test case.

**Definition 15**

Let \( AP_{n} \) be the coverage criterion \((C)\) "all paths of length \( \leq n \)." The quality for the criterion \( AP_{n} \) is the following:

\[
q_{AP_{\leq n}} = 1 - \left(1 - \frac{1}{|\mathcal{P}_{\leq n}|}\right)^N
\]

Given the desired quality \( q_{AP_{\leq n}} \) and the number of paths \( P_{\leq n} \), we can compute the required number of test cases as:

\[
N \geq \frac{\log(1 - q_{AP_{\leq n}})}{\log(1 - \frac{1}{|\mathcal{P}_{\leq n}|})}
\]

When conducted for the criterion \( AP_{n} \), the original GMST-SP method consists of the following steps:

1) Construct the CFG for a given program.
2) Compute the numbers \( f_{v}(i) \) for every \( 0 \leq i \leq n \) and every vertex \( v \) of the CFG by using the recurrence rules given above.
3) Compute the required number of test cases \( N \) for the given requested testing quality \( q \) and the number of paths whose length is less or equal to \( n \).
4) Generate \( N \) paths uniformly (this means that all the paths have the same probability).
5) Generate input data for the generated paths.
6) Execute \( N \) test cases driven by the generated input data.

**GMST method for task trees**

**Definition 16**

Let \( \{ T_1, T_2, ..., T_k \} \) be a task tree family, let \( T_i \) be the \( i^{th} \) task tree
In this paper, we classify a fault as a visible fault or a deeply hidden fault. Generally, due to non-deterministic execution, a bug in a parallel program may not be detected even when the program is executed multiple times (Yang, 1990). Therefore, in this paper, we are targeting hidden faults and deeply hidden faults in particular. A fault is hidden in a path if it does not expose itself during the first traversal of the path. The hiding depth \( d \) of a hidden fault is the number of path traversals needed to reveal the hidden fault. We classify a fault as a visible fault if it's \( d=1 \) or as a hidden fault if it's \( d>1 \). A fault is a deeply hidden fault if it's \( d>>1 \). Interpretation of the condition \( d>>1 \) depends on the application at hand. For example, in our study, we interpret \( d>>1 \) as \( d=10 \).

Moreover, we are especially interested in revealing deeply hidden faults that are hidden in the most frequently executed paths on the target platform. The motivation for this interest is the fact that the absence of faults in seldom executed paths does not improve the product reliability too much, but the presence of faults in the most frequently executed paths decreases drastically the product reliability.

### Experimental setup

The experiments were conducted by measuring the testing effort (time), the path coverage, and the fault-revealing capability of each of the three presented methods. The tested objects were the TTE task tree families that were constructed from three, four, five, six, and seven tasks. A deeply hidden fault with the depth ten \((d=10)\) was seeded in the most frequent path for each task tree within the family. The most frequent paths were previously determined based on the statistics of native parallel execution of each of the task trees on the target platform, which was the dual-core symmetric multiprocessor, Intel® Core(TM) i5 CPU M 520 @ 2.4 GHz, 4 GB RAM, with Windows7 Professional® 64-bit OS.

### RESULTS

#### Testing effort

Table 1 shows testing effort (execution time in seconds) for the three methods earlier described. The columns of Table 1 are as follows:

1) The columns No tasks, No trees, and No paths show related number of tasks, trees and paths, respectively.
2) The columns ET time, GMST time, and SUT time show execution time needed to run ET, GMST and SUT test cases, respectively. The column GMST time is partitioned into three sub columns for three distinctive values of required quality \( q \) (0.9, 0.95 and 0.99). Similarly, the column SUT time is divided into three sub columns for three different values of reliability \( r \) (0.9, 0.95, and 0.99; in all three cases \( M = 1.4\% \)).

#### Targeted fault types

Generally, due to non-deterministic execution a bug in a parallel program may not be detected even when the program is executed multiple times (Yang, 1990). Therefore, in this paper, we are targeting hidden faults and deeply hidden faults in particular. A fault is hidden in a path if it does not expose itself during the first traversal of the path. The hiding depth \( d \) of a hidden fault is the number of path traversals needed to reveal the hidden fault. We classify a fault as a visible fault if \( d=1 \) or as a hidden fault if \( d>1 \). A fault is a deeply hidden fault if it's \( d>>1 \).

### Testing coverage

Table 2 shows the number of uncovered paths for the

<table>
<thead>
<tr>
<th>No tasks</th>
<th>No trees</th>
<th>No paths</th>
<th>ET time (s)</th>
<th>GMST time (s)</th>
<th>SUT time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>q=0.9</td>
<td>q=0.95</td>
<td>q=0.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>r=0.9</td>
<td>r=0.95</td>
<td>r=0.99</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>18</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>180</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>120</td>
<td>2700</td>
<td>18</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>720</td>
<td>56700</td>
<td>230</td>
<td>534</td>
<td>605</td>
</tr>
</tbody>
</table>

GMST method for task trees consists of the following steps:

1. Generate the task tree family (using Algorithms 1 to 3).
2. For each task tree \( T_i \) within the generated task tree family do step 2a through 2c.
3. Create the evolution graph (using Algorithm 4).
4. Collect all the paths (using Algorithms 5 to 7).
5. Set \( h \), the number of paths in \( T_i \).
6. Compute \( N = \sum h \).
7. Compute \( q \) (note that \( H = P_{sd} \)).
8. For each one of \( N \) test cases do step 5a through 5d.
9. Generate a random number \( r \).
10. Determine the corresponding task tree number \( i \) and the path number \( j \) using Algorithm 8.
11. Convert the path \( z_j \) into the corresponding task schedule \( s_i \).
12. Execute the schedule \( s_i \) on the target OS simulator.

Experimental setup

The experiments were conducted by measuring the testing effort (time), the path coverage, and the fault-revealing capability of each of the three presented methods. The tested objects were the TTE task tree families that were constructed from three, four, five, six, and seven tasks. A deeply hidden fault with the depth ten \((d=10)\) was seeded in the most frequent path for each task tree within the family. The most frequent paths were previously determined based on the statistics of native parallel execution of each of the task trees on the target platform, which was the dual-core symmetric multiprocessor, Intel® Core(TM) i5 CPU M 520 @ 2.4 GHz, 4 GB RAM, with Windows7 Professional® 64-bit OS.
Table 2. The number of uncovered paths.

<table>
<thead>
<tr>
<th>No task</th>
<th>No trees</th>
<th>No paths</th>
<th>ET time (s)</th>
<th>GMST time (s)</th>
<th>SUT time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>q=0.9</td>
<td>q=0.95</td>
<td>q=0.99</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>18</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>180</td>
<td>0</td>
<td>17</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>120</td>
<td>2700</td>
<td>0</td>
<td>276</td>
<td>120</td>
</tr>
<tr>
<td>7</td>
<td>720</td>
<td>56700</td>
<td>0</td>
<td>5588</td>
<td>2796</td>
</tr>
</tbody>
</table>

Table 3. The mean path coverage depth.

<table>
<thead>
<tr>
<th>No tasks</th>
<th>No trees</th>
<th>No paths</th>
<th>ET time (s)</th>
<th>GMST time (s)</th>
<th>SUT time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>q=0.9</td>
<td>q=0.95</td>
<td>q=0.99</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>18</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>180</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>120</td>
<td>2700</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>720</td>
<td>56700</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

three methods earlier described. The columns of Table 2 are as follows:

1) The columns No tasks, No trees and No paths are the same as in Table 1.
2) The columns ET, GMST and SUT show the number of uncovered paths. The latter two columns are partitioned into three sub columns for various values of the desired quality \( q \) and reliability \( r \), equally as in Table 1.

Table 3 shows the mean path coverage depth for the three methods in question. Table 3 is organized equally as Table 2.

DISCUSSION

Testing effort

The results in Table 1 show exponential growth of ET testing time with the number of tasks used to construct a task tree family. The GMST testing time shows this exponential growth too, and is even greater than ET testing time, because GMST method requires more test cases than ET method. On the contrary, SUT testing time shows logarithmic growth with the number of tasks; it starts with a greater value, but increases rather slowly, and later becomes less than GMST time and ET time (for \( r \) less than 0.99).

These trends become more obvious after having a look into Figure 2, which illustrates logarithm base 10 of the test execution time as a function of the number of tasks for ET method, GMST method (for \( q=0.95 \)), and SUT method (for \( r=0.95 \)). A reader should be aware of a minor discrepancy between Table 1 and Figure 2 that was introduced by replacing zeroes with ones in Table 1 in order to enable plotting the curves (because \( \log_{10} 0 \) is undefined).

It should be obvious from Table 1 and Figure 2 that SUT is scalable with the number of tasks, whereas ET and GMST methods are not scalable. Practically, SUT can be used even on normal desktop and laptop PCs. On the other hand, application of ET and GMST requires longer execution times on more powerful servers that are typically available for mission and safety critical applications.

Testing coverage

The results in Table 2 show that only ET method provides complete coverage of all the paths, since number of uncovered paths is zero for all the task tree families. GMST method provides partial coverage of paths, where the number of uncovered paths increases exponentially with the number of tasks. The number of uncovered paths for SUT method also shows the same exponential growth as the number of uncovered paths for GMST method, but is even an order of magnitude greater than the number of uncovered paths for GMST method.

Figure 3 makes these trend obvious by illustrating the logarithm base 10 of the number of uncovered paths as a function of the number of tasks for ET method, GMST method (for \( q=0.95 \)), and SUT method (for \( r=0.95 \)). A reader should be aware that all zeros in Table 2 were
replaced with once to enable plotting logarithmic curves.

As Figure 3 shows ET, it provides total paths coverage, but a reader should note that this coverage is extremely shallow, each path is covered just once. Therefore, ET can discover only unhidden faults (faults at depth 1). On the other hand, partial coverage of paths in case of GMST and SUT methods is explained by randomness they use in selecting paths. This randomness is not chaotic. On the contrary, it is driven by a meaningful intention to favor more frequent/probable paths. Relatively poor overall coverage for SUT method is a consequence of SUT method nature – it favors the most frequent paths and neglects infrequent paths, but when testing is based on product’s operational profiles that is exactly the main goal of testing. GMST provides better overall paths coverage than SUT by uniformly favoring more probable paths over less probable paths for the price of increased testing effort.

The main GMST advantage over SUT is that GMST’s paths coverage is uniform, whereas the SUT’s paths coverage is non-uniform. Although, this advantage may be recognized by intuition, based on the nature of GMST
and SUT, it becomes apparent only after analyzing data presented in Table 3.

The results in Table 3 show that ET method has the smallest mean path coverage depth, which is always equal to one, because each path is covered once and once only. SUT method has greater mean path coverage depth than ET, which increases with reliability \( r \), but exponentially decreases with the number of tasks. Therefore, ET’s mean paths coverage is not scalable. On the other hand, GMST method also has greater mean path coverage depth than ET, which also increases with the required quality \( q \), but the main quality of GMST’s mean paths coverage is that it is constant in respect to the number of tasks, so it is scalable.

Figure 4 shows the logarithm base 10 of the mean path coverage depth as a function of the number of tasks for ET method, GMST method (for \( q=0.95 \)) and SUT method (for \( r=0.95 \)). Replacing zeros with ones in Table 3 to enable plotting curves now also becomes meaningful, because these curves may be interpreted as levels of deep path coverage. Since ET cannot discover hidden faults, the level of its deep path coverage is zero, and indeed logarithm base 10 of ET’s mean paths coverage (which is 1) is zero. SUT’s curve is a steep decreasing line showing its non-scalability. GMST’s curve is a horizontal line that proves scalability in respect to the number of tasks and also shows the level of deep path coverage. Actually, Table 3 shows that GMST’s mean paths coverage for \( q=0.95 \) is 3, which means that GMST is able to discover hidden faults at the depth 3.

Even more importantly, the level of GMST’s deep paths coverage is regulated very easily just by changing desired value of \( q \). For example, Table 3 shows that by increasing \( q \) to the value 0.99 we increase the level of deep paths coverage (the corresponding horizontal line in Figure 4 would shift up) such that the mean paths coverage becomes 5.

Conclusions

ET method does not provide deep path coverage since all the paths are covered exactly once rather than multiple times, and it also requires great testing effort. GMST and SUT both have better deep path coverage than ET. SUT requires less testing effort than GMST and ET, but its deep path coverage decreases with the number of tasks. Finally, GMST has advantage over SUT because it provides constant mean level of deep path coverage, which does not depend on the number of tasks, and which can be regulated by the required testing quality \( q \), such that it becomes deeper with the increase of required quality \( q \). The price for this advantage of GMST over SUT is greater testing effort, which might be considered tolerable for mission-critical projects.

Finally, we recommend:

1) Usage of ET method in case when the target is to discover only unhidden faults.
2) Usage of SUT method in case when limited testing effort is affordable and when the goal of testing campaign is to reveal deeply hidden faults only on the most frequently executed paths.
3) Usage of GMST method in case when more testing effort is affordable and when the goal of testing campaign is to reveal hidden faults on the most probable paths and
with the uniform path coverage depth.

ACKNOWLEDGEMENTS

This work was partly supported by the Serbian Ministry of Science and Technology under Grants TR32031 and III44009.

REFERENCES


