2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)

Improving the Detection of Sequential Anomalies Associated with a Loop

Faisal Fahmi*, Pei-Shu Huang**, Feng-Jian Wang**

* EECS Int'l Graduate Program
National Chiao-Tung University, Taiwan
faisalfahmiy@gmail.com

** Dept. of Computer Science
National Chiao-Tung University, Taiwan
{pshuang, fjwang}@cs.nctu.edu.tw

***Abstract*— Workflow models are widely applied in business software design. A workflow model contains a set of systematic ordered tasks to achieve designated business goal(s) under the designed flow control. Analyzing artifact usage during design phase can prevent unexpected artifact result due to abnormal artifact operation(s). A sequential anomaly indicates a pair of activities operating on the same artifact that can result in redundant write or missing production. On the other hand, the iteration of a loop structure in a workflow cannot be statically analyzed, thus, detecting process of artifact anomalies in a loop is costly. In this paper, we present an effective method to detect all anomalies associated with a loop by removing the redundant computation due to the repeated structure of the body and control in the iterations. After the removing, the anomalies can be detected on a single iteration generated instead. Here, the process of anomaly detection is now simplified into two phases: First, a workflow model is transformed into a corresponding C-tree structure and next, the proposed anomaly detection methodology is applied to the C-tree. Compared with current approaches, our method can reduce the space complexity and decrease the execution times of anomaly detection as linear.**

Keywords— artifact anomaly, workflow, redundant computation

I. INTRODUCTION

Workflows are used to define business processes within enterprises, and Workflow Management Systems (WMS) automate the processes by completing tasks which realize parts of business goals in a designated order [1]. Analysis of a workflow is essential to guarantee the correctness of its execution. To improve workflow, method in [2] detects inconsistent dependencies among tasks to assure the safety of a workflow. Van der Aalst et al. developed effective Petri-net based techniques to verify deadlocks, livelocks (infinite loops), and dead tasks from workflow schemas [3][4][5][6].

Besides the activities inside workflow models, the artifacts (i.e., the data or objects defined, used, or referenced by the activities) are also essential to workflow execution. Conflicts among the artifacts may cause crashes during enactment of workflows even the workflows are all well behaved. The conflicts among operations on artifacts have been studied, and a series of results based on both structural and temporal factors were presented [7][8][9].

[10] presented a C-tree structure for detecting the anomalies inside a structured workflow. This work shows the two activity nodes which work on the same artifact(s) sequentially and whose operation are abnormal. Moreover, the methods for detecting these anomalies need time complexity O(n), faster than the other methods [11][12][13].

In this paper, we present an effective method to detect all sequential anomalies associated with a loop by removing the redundant computation due to the repeated structure of the body and control in the iterations. Our methodology adopts a C-tree structure (defined in Section III) which increases the convenience and elegancy of anomaly detection and helps lower the size of nodes to be analyzed while not losing the abilities of detecting anomalies within workflow models. Comparing with [10], our method can reduce the space complexity, although it cannot have better execution time. Here, the process of anomaly detection is now simplified into two phases: First, a workflow model is transformed into a corresponding C-tree structure and next, the proposed anomaly detection methodology is applied to the C-tree.

The rest of this paper is organized as following: In Section 2, the background and related work of workflow analysis are introduced. In Section 3, we introduce a C-tree used for the analysis. In Section 4 based on the C-Tree, we propose the sequential anomaly detection method. In Section 5, we discuss about the efficiency of anomaly detection method.

II. BACKGROUND

A. Definitions of Structured Workflow

Workflow models are widely applied in business software design. A workflow model contains a set of systematic ordered tasks to achieve designated business goal(s) under the designed flow control. A workflow can be defined as a directed graph where a node represents an activity and edges represents the flows between nodes [3]. Besides activity nodes, the nodes indicating the control flows are called control nodes. A control node splits execution of a workflow under designated conditions, or joins the processes from different execution paths together. Analyzing the workflow can be complicated if the splits and joins among control nodes are not constrained. Kiepuszewski et al. defined the structured workflow model (SW) in [14], and claimed that SW with well-developed modeling environment is helpful in preventing deadlock or unwanted multiple instances. In an SW, each split node has a corresponding joint node at the same level, and the nodes inside the split-joint structure have no edges crossing to the nodes outside the split-joint structure. In this paper, the control structures applied in SW include AND, XOR, and Loop.

B. Definition of Anomaly

Common workflow anomalies include deadlock, infinite loop, lack of synchronization, and dangling reference [5][15][16]. In [10], Wang et al., defines a data flow anomaly based on the abnormal usages of an artifact as followings: (1) Redundant Write happens when an artifact is defined (written) by an operation but the artifact is required by neither the succeeding operations nor a member of the process outputs. (2) Conflict Write means accesses among artifacts are conflicted in one of the following ways: multiple parallel productions, multiple parallel updates, or parallel read and update. (3) Missing Production occurs when an artifact is consumed before it is produced or after it is destroyed. The anomalies defined in [10] are further divided into two categories, sequential and parallel anomalies [17], where the former includes redundant write and missing production, and the latter includes conflict write.

978-1-7281-2607-4/19/$31.00 ©2019 IEEE
DOI 10.1109/COMPSAC.2019.10195

In this paper, we focus on a sequential anomaly which is one of the major concerning issues in workflow analysis since it can occur in both sequential and parallel structures. To detect the anomalies, the analysis includes two operations for the same artifact. From the data access point of view, an operation of artifact can be regarded as one of the following operations: Read, Write, or Kill [10]. To simplify the analysis, a sequential anomaly is detected when two sequential operations working on the same artifact in one of the following ways: (Kill, Read), (Kill, Kill), (Write, Kill), and (Write, Write) [17].

C. Related Work on Workflow Analysis

Sadiq et al. presented a visual verification approach in [18] employing a set of graph reduction rules to identify structural conflicts in the given workflow and can identify deadlock and lack of synchronization conflicts. In [19], Van der Aalst et al. reviewed soundness notions and analysis approaches of workflow, and show the theoretical limits of workflow verification. The author compared different notions of verification and concluded that many errors can be discovered using reduction rules. Li et al. [20] analyzed resource constraints of a workflow specification with timed specifications. They worked at static workflow, but lack of dynamic analysis on workflows.

III. Generating A C-tree For Analysis

The SW can be transformed into a different structure, called as a C-Tree [21], to detect the sequential anomalies. Section III.A shows the definition of a C-Tree and Section III.B shows that an SW is transformed into a C-Tree with separation of sequential and parallel concepts.

A. Definition of a C-tree

A C-Tree proposed in [21] is defined to represent the structured workflow. Each node X in a C-tree has a distinct corresponding node Y in an SW graph and the C-tree node contents are shown in Definition 3.1: A left_child and parent pointers indicate the address of C-tree's node corresponding to Y's successor and predecessor in SW, respectively. The left_child pointer is null if Y has no successor in SW. In a C-Tree, a split-joint structure in SW is represented by a split node and right_branches indicating the address set of C-Tree's nodes, of which each represents the first node of a distinct branch inside the structure in SW. If X's node-type is an ACTIVITY, START or END, the value of right_branches is null. Besides, an operation_list is a linked list used to represent the operation sequence of artifact(s) in a node. Each element in the operation_list is an artifact_operation that contains an artifact (artifact_id) and its operation (operation_type).

Definition 3.1. A Concurrent-Tree (C-tree) node

```
1  List<string> node_types = {"START", "END",
   "ACTIVITY", "AND", "XOR", "LOOP"}
2  List<string> loop_types = {"NONE", "WHILE", "FOR",
   "REPEAT"}
3  List<string> operation_types = {"READ", "WRITE",
   "KILL"}
4  artifact_operation{
5      string artifact_id;
6      operation_types operation_type;
7  }
8  ctree_node {
9      string node_id;
10     node_types node_type;
11     loop_types loop_type;
12     int level;
13     ctree_node * left_child;
14     ctree_node * parent;
15     ctree_node * right_branches {};
16     LinkedList<artifact_operation> operation_list {};
17 }
```

B. Transforming an SW into a C-tree

A node in SW can be a simple or complex node, where a simple node is an atomic node and complex node represents a structural node which may contain two or more simple nodes. Based on definition 3.1, an SW can be transformed into a C-tree by applying algorithm 3.1. The algorithm traverses each simple node in the input SW completely from its starting node. The switch statement in Line 3 works to generate a C-tree node based on the type of current SW node. Lines 4~7 handle the start and end nodes. Lines 8~16 handle a joint node. All joint nodes, such as an AND joint node, have no correspondent node in a C-tree and introduce nothing. Lines 17~25 handle the split nodes such as AND, XOR and LOOP. The left_child of a split node, called split_successor, is used to represent the successor of joint node in SW, e.g., node E in shown Fig. 1 is split_succesor of LOOP split. To simplify the anomaly detection in an XOR node with empty path, an empty ACTIVITY node is generated to represent an empty path, e.g., F node shown in Fig. 1 represents the empty path of XOR branch in SW. Line 26~30 handle an activity node. An artifact_operation at Line 28 is defined in Definition 3.1, e.g., artifact_operation(s) of node B shown in Fig. 1 are (V1, READ), (V2, WRITE), and (V3, WRITE). Fig. 1 shows an example an SW and its C-Tree generated by Algorithm 3.1.

In Algorithm 3.1, the arc of a node with node-type ACTIVITY or JOINT indicates the immediate successor of the node in SW. Since a joint node in SW has no corresponding node in a C-tree, the immediate successor of a joint node indicates the immediate successor of its corresponding split nodes in a C-tree. Besides, for a split node, the arc(s) indicates the ith successor of the split node in SW.

Algorithm 3.1 GTT (Graph to C-tree)
Global variable: SW Node * split_successor, int level
Input: SW Node * p, ctree_node * parent
Output: ctree_node *node

```
1  ctree_node *node = new ctree_node(p, parent);
2  node->parent = *parent;
3  switch (p) {
4    case ("START")
5      node->left_child = GTT(p's immediate successor,
   node);
6    case ("END")
7      node->left_child = NULL;
8    case ("JOINT")
9      if (p's immediate successor is not a JOINT)
10       split_successor = p's immediate successor;
11     else
12       split_successor = NULL;
13     if (parent is XOR)
14       new empty ACTIVITY ctree_node;
15     node->parent = NULL;
16     node-> level = level - 1;
17   case ("SPLIT")
18     node-> level = level + 1;
19     for (int i=0; i<b; i++){ //b is the number of branches in p
20       ctree_node *right_branch=GTT(p's i^th successor, node);
21       if (right_branch is not NULL)
22         node-> right_branches.add(right_branch);
```

```
23      } //end of for
24      if (split_successor is not NULL)
25        node->left_child = GTT(split_successor, node);
26    case ("ACTIVITY")
27      node->level = level;
28      for each artifact_operation a in p
29          node->operation_list.add(a);
30      node->left_child = GTT(p's immediate successor, node);
31  }
32  return node;
```

Algorithm 3.1 traverses the input SW from the starting node as a root, and generate corresponding C-Tree structure. Since each node in the SW is visited at most once, the time complexity of the transformation is O(n), where n represents the total number of nodes in the input workflow model.

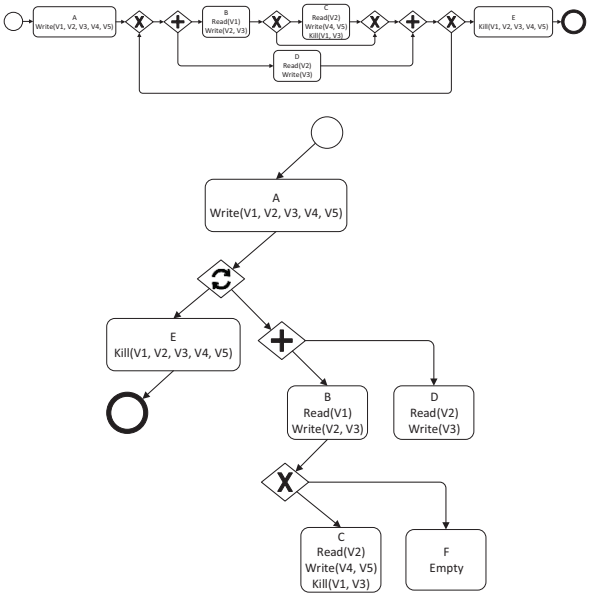

Fig. 1: An example of an SW and its generated C-tree.

IV. Detection of Sequential Anomaly Assosiated with A Loop

In an SW, a loop can be structured as a sequence of loop body and control, where a loop body can contain sequential or parallel structures. In this paper, we discuss anomaly detection from the viewpoint of loop and present a method which can detect all sequential anomalies below in an SW.

A sequential anomaly, called SNA, can be represented as a three-Tuple (x, p, q) where an artifact x is operated by two operations p and q, where p is performed before q and there is no operation on x between p and q. As a result, there are four types of sequential anomalies shown in definition 4.1.

Definition 4.1. Sequential Anomaly (SNA)

SNA1={(x, p, q) | x is killed on p and used (read) on q and ∃ a path from p to q & no operation for x in the path}

SNA2={(x, p, q) | x is killed on p and q and ∃ a path from p to q & no operation for x in the path}

SNA3={(x, p, q) | x is defined (write) on p and killed on q and ∃ a path from p to q & no operation for x in the path}

SNA4={(x, p, q) | x is defined (write) on p and q and ∃ a path from p to q & no operation for x in the path}

Based on Definition 4.1, to perform anomaly detection for an artifact A in a node N, the data of A's latest operation which is Write or Kill before reaching N and A's first operation in N, are needed. To simplify the discussion, the set of first and last operations of artifact(s) in a node is defined as a series of SA sets in section IV.A. The set of latest operation of artifact(s) which is Write and Kill before reaching a node is defined as LK and LW, respectively, in Section IV.B. Section IV.C describes Algorithm 4.2~4.6 used to traverse the C-tree to detect the sequential anomalies. Section IV.D shows an example of anomaly detection based on the algorithms.

A. The Sets of First and Last Operation in a Node (SA Sets)

Each element of an SA sets defined below is a couple (V, N), where an artifact V may have an operation run inside a simple node N. In N, each of the following series of SA sets is defined to contain the artifact(s), which is operated with the first/last operation of the corresponding type.

1. SAFR(N): The set of artifacts, whose first operation is READ in N.
2. SAFW(N)/SALW(N): The set of artifacts, whose first/last operation is WRITE in N respectively.
3. SAFK(N)/SALK(N): The set of artifacts, whose first/last operation is KILL in N respectively.

For a C-tree node, Algorithm 4.1 derives the SA sets from operation_list of the node. For an operation_list of the input node, Algorithm 4.1 handles each artifact_operation element by element. An LO set at Line 2 is used to help put an artifact_operation being handled into a corresponding SA set. During the for loop in Lines 3~22, LO contains a set of artifact_operation(s), where each element (artifact_id, operation) represents the lastest operation of the artifact (artifact_id). There are four functions defined to process LO set: 1) GetElement at line 4 gets the current artifact_operation called as operated_ao. Find at line 6 examines whether the artifact_id in the operated_ao exists in the LO set. When the artifact exists, it returns true. False, otherwise. If the return value is true, the function UpdateLastOperation will be executed, otherwise, PutLastOperation. 2) PutLastOperation at line 9 puts the operated_ao into the LO set. 3) UpdateLastOperation at line 7 replaces the artifact_operation in LO set with operated_ao if the artifact_id is the same. The switch statement at Lines 10~20 derive the sets of first operations of artifact(s) in the input node, such as SAFR, SAFK, and SAFW. Besides, Lines 23~29 work to derive the sets of last operation of artifact(s) in the input node, such as SALK and SALW.

Algorithm 4.1 Derive a series of SA sets

Input: ctree_node *node *// a ctree_node is generated by Algorithm 3.1*

Operation: derive a series of SA sets of the input node

```
1   LinkedList<artifact_operation> op_list = node->
    operation_list;
2   List<artifact_operation> LO {};
3   for (int i=0; i<op_list.size; i++) {
4     artifact_operation operated_ao =
    GetElement(op_list, i);
5     SA_element e = new SA_element(operated_ao->
    artifact_id, node-> node_id);
6     if(Find(LO, operated_ao)) {
7         UpdateLastOperation(LO, operated_ao);
8     } else {
9         PutLastOperation(LO, operated_ao);
```

```
10          switch(operated_ao->operation) {
11          case ("READ"):
12                node->SAFR.Put(e);
13            break;
14          case ("WRITE"):
15                node->SAFW.Put(e);
16            break;
17          case("KILL"):
18               node->SAFK.Put(e);
19            break;
20          } //end of switch
21    } //end of if
22  } //end of for
23  for each artifact_operation lo_ao in LO {
24    SA_element e = new SA_element(lo_ao->
    artifact_id, node->node_id);
25    if(lo_ao->operation==="WRITE")
26       node->SALW.Put(e);
27    else if(lo_ao->operation==="KILL")
28       node->SALK.Put(e);
29  }
```

B. *The Sets of Lastest Operation*

To simplify the computation, the set of artifact(s) whose latest operation may be Kill/Write before reaching node X can be used to define LK(X)/LW(X) respectively. In other word, each element (V, N) in LK(X)/LW(X) indicates that there is a path of no operation for artifact V from node N to X and V is an element of SALK(N)/SALW(N), respectively. Given a path from the starting node of the whole program to the ending of node N, the path is called as Start-of-program-to-End-Path of N, SEP(N). Thus, SEP(N) may have more than one execution path. The set of artifact(s) whose latest operation is Kill/Write in SEP(N) can be used to define NLK(N)/NLW(N), respectively, where each element (V, Q) in NLK(N)/NLW(N) indicates there is an SEP(N) which has no operation for artifact V after node Q, and V is an element of SALK(Q)/SALW(Q), respectively.

An SW can contain activity node(s) and control structure(s). A simple activity node can contain a sequence of operation(s), thus, the ending of the node in SEP can be defined as the ending of this sequence of operation(s). A simple control structure contains a split node and its corresponding joint node at the same level, where the other node(s) inside the structure have no edges pointing outside the structure. Thus, the ending node of a control structure in SEP is its joint node.

Beside sequential structure, there are two types of control structure discussed above: (1) loop-less control structure and (2) loop control structure, where the former includes XOR and AND structures and the latter includes repeat, for, and while loops at least. For a loop-less control structure, its SEP is a path from the starting node of the whole program to the ending of joint node of the structure and its immediate successor can be deemed as the immediate successor of its joint node. For a loop control structure, its SEP is a path from the starting node of the whole program to the ending of joint node of the structure, where the loop body and control are executed in n iterations, n is maximum number of iterations, and its immediate successor can be deemed as the immediate successor, outside the structure, of its join node after n iterations of loop body and control. Based on above definitions, the scheme for calculation of LK/LW and NLK/NLW are defined as follows.

Scheme 1. LK/LW and NLK/NLW in an SW.

1. For two simple activity nodes A and B in an SW, if B is an immediate successor of A, NLK(A) and NLW(A) are the same as LK(B) and LW(B), respectively.
2. For two simple loop-less control structures K and L in an SW, if L is an immediate successor of K, NLK(K) and NLW(K) are the same as LK(L) and LW(L), respectively.
3. For two simple loop control structures P and Q in an SW, if NLK(P) and NLW(P) calculations include n iteration of loop body and control, and Q is an immediate successor of P, NLK(P) and NLW(P) are the same as LK(Q) and LW(Q), respectively.

For node A in scheme 1, NLK(A) and NLW(A) contain the set of artifact(s) whose latest operation is Kill/Write in a path from the starting node of the whole program to the ending of node A, respectively. On the other hand, LK(B) and LW(B) contain the set of artifact(s) whose latest operation may be Kill/Write before reaching node B. If B is the immediate successor of A, there is an empty path between A and B. Thus, obviously, NLK(A) and NLW(A) are the same as LK(B) and LW(B). For nodes K and L and P and Q in scheme 1, the proof is similar to the above for nodes A and B above, respectively [21].

C. *Sequential Anomaly Detection*

In subsection II.B, a sequential anomaly is detected when two sequential operations working on the same artifact in one of the following ways: (Kill, Read), (Kill, Kill), (Write, Kill), and (Write, Write) [17]. Consider a C-tree node N, the 1st operation of the anomaly can be found in LK(N) or LW(N) and the 2nd operation of the anomaly can be found in SAFR(N), SAFK(N), or SAFW(N). For a loop, the number of iterations cannot be specified statically, and, the analysis in a C-tree of loop need be done with extra efforts. Subsection C.1 discusses the anomaly detection in a loop-less C-tree structure and Subsection C.2 shows that a loop structure in a C-tree can be simplified for analysis.

1) *Sequential anomaly in a loop-less structure*

For a *loop-less* C-tree structure, an O(n) sequential anomaly detection method can be developed. The method adopts a mutual recursive to traverse the C-tree, where Algorithm 4.2 is its starting method. The algorithm calls the *detection routine* based on the node_type of current node in the C-tree, i.e., START, END, XOR, AND, and ACTIVITY. When it handles START node, *PutAllArtifact* at Line 3 puts all artifacts in the C-Tree into the LK of the starting node. This LK will be useful to detect anomaly of SNA1 defined in definition 4.1, i.e., an artifact is not defined but accessed. When it handles END node, it replies the NULL value representing the ending of detection. For ACTIVITY node, *ProcessActivityNode* at Line 8 detects the anomaly inside the current node directly. For XOR and AND nodes, *ProcessXorNode* at Line 10 and *ProcessAndNode* at Line 13 detect the anomalies inside each right branch of the corresponding node recursively. An *and_oa* set at Line 12 is used to help calculate the NLK and NLW of an AND structure in a C-tree. Each element (*artifact_id*, *level*) in *and_oa* indicates the artifact (*artifact_id*) is operated in one or more right branch(es) of AND structure at corresponding *level* in SW. The *level* is used to identify the corresponding AND

structure since an AND structure may contain another AND structure. After detection completes in the routine called, it examines the existence of left_child. If the left_child exists, it handles left_child recursively. Otherwise, it replies the data flow used for analysis correspondingly.

Algorithm 4.2 SAD (Sequential Anomaly Detection)
Input: ctree_node *node
Output: ctree_node *node
Global variables: SNA1, SNA2, SNA3, SNA4, and_oa

```
1   switch(node->node_type) {
2       case("START"):
3           PutAllArtifact(node->LK);
4           return SAD(node->left_child);
5       case("END");
6           return node = NULL;
7       case("ACTIVITY");
8           return ProcessActivityNode(node);
9       case("XOR"):
10          return ProcessXorNode(node);
11      case("AND");
12          List<string artifact_id, int level> and_oa;
13          return ProcessAndNode(node);
14  }
```

The detection routine ProcessActivityNode, shown in Algorithm 4.3, detects the anomaly in an activity node based on LK, LW, SAFR, SAFK, and SAFW. Line 1 generates the SA sets of the node. Lines 2 and 3 derive the LK and LW from the NLK and NLW of its parent node, respectively (Scheme 1). Lines 4~7 detect the anomalies SNA 1~4 correspondingly. Lines 8 and 9 calculate the NLK and NLW of the node based on the LK and LW, and SA sets respectively. A function removeFirstSet is used to remove an element of LK or LW if the artifact_id of the element exists in one of SAFR, SAFW, and SAFK. Since an activity node can be inside a right branch of AND structure in a C-tree, Lines 10~15 are used to update the and_oa set. A for loop at Line 11 examines each artifact_id in the artifact_operation of the node. If the artifact_id doest not exist in and_oa, it will be put in and_oa with its corresponding level of node in C-tree. Otherwise, it will be ignored. Lines 16~19 handle the mutual recursive routine by checking the existence of left_child. If the left_child exists, it handles left_child recursively. Otherwise, it replies the data flow used for analysis correspondingly.

Algorithm 4.3 ProcessActivityNode
Input: ctree_node *node
Output: ctree_node *node *// return the data flow*
Global variables: SNA1, SNA2, SNA3, SNA4, and_oa

```
1   processSAsets(node);
2   node->LK = node->parent->NLK;
3   node->LW = node->parent->NLW;
4   SNA1 += checkSeqAnomaly(node->LK, node->SAFR);
5   SNA2 += checkSeqAnomaly(node->LK, node->SAFK);
6   SNA3 += checkSeqAnomaly(node->LW, node->SAFK);
7   SNA4 += checkSeqAnomaly(node->LW, node-> SAFW);
8   node->NLK = removeFirstSet(node->LK, (node->SAFR ∪
            node->SAFW ∪ node->SAFK)) ∪ node->SALK;
9   node->NLW = removeFirstSet(node->LW, (node->SAFR ∪
            node->SAFW ∪ node->SAFK)) ∪ node->SALW;
10  if(node->level > 0){
11      for each artifact_operation ao in node{
12        if (isExist(and_oa, ao->artifact_id))
13          putOaElement(and_oa, ao->artifact_id, node-> level)
14      }
15  }
16  if (node->left_child != null)
17      SAD(node->left_child);
18  else
19      return node;
```

The detection routine *ProcessXorNode* is shown in Algorithm 4.4. An XOR structure in SW contains two parts: (1) the expression, and (2) the branches. The expression part inside an XOR structure is referred to artifact(s) in general. As a result, it may have the anomaly due to the operation in the previous nodes. In a C-tree, the expression of an XOR structure in the corresponding SW is represented as XOR node itself. For the part (1), Line 1 generates the SA sets of the node. Lines 2 and 3 derive the LK and LW from the NLK and NLW of its parent node, respectively (Scheme 1). Lines 4~7 detect the anomalies SNA 1~4 for the expression of the XOR node correspondingly. There are two NLKs and NLWs in the algorithm: 1) NLK and LNW in Lines 8 and 9 indicates the NLK and NLW of the XOR split node in SW, where they will be used to detect the anomalies inside the branches of XOR. 2) NLK and NLW in Lines 16 and 17 indicates the NLK and NLW of the XOR joint node in SW, where they will be used to detect the anomalies in left_child of the node in C-tree, i.e., a successor node of XOR structure in SW. For each branches of the part (2), Lines 10~15 perform anomaly detection and collect the data flow by mutual recursive callings. Lines 18~21 handle the mutual recursive routine by checking the existence of left_child. If the left_child exists, it handles left_child recursively. Otherwise, it replies the data flow used for analysis correspondingly.

Algorithm 4.4 ProcessXorNode
Input: ctree_node *node
Output: ctree_node *node *// return the data flow*
Global variables: SNA1, SNA2, SNA3, SNA4, and_oa

```
1   processSAsets(node);
2   node->LK = node->parent->NLK;
3   node->LW = node->parent->NLW;
4   SNA1 += checkSeqAnomaly(node->LK, node->SAFR);
5   SNA2 += checkSeqAnomaly(node->LK, node->SAFK);
6   SNA3 += checkSeqAnomaly(node->LW, node->SAFK);
7   SNA4 += checkSeqAnomaly(node->LW, node->SAFW);
8   node->NLK = removeFirstSet(node->LK, (node->SAFR
            ∪ node->SAFW ∪ node->SAFK)) ∪ node->SALK;
9   node->NLW = removeFirstSet(node->LW, (node->SAFR
            ∪ node->SAFW ∪ node->SAFK)) ∪ node->SALW;
10  ctree_node *temp_branch;
11  ctree_node *branch = new ctree_node[node->right_
        branches.size];
12  for(int i=0; i < node->right_branches.size;i++){
13      branch[i] = SAD(right_branches[i]);
14      temp_branch = temp_branch ∪ branch[i];
15  }
16  node->NLK = temp_branch->NLK;
17  node->NLW = temp_branch->NLW;
18  if (node->left_child != null)
19      SAD(node->left_child);
20  else
21      return node;
```

The detection routine *ProcessAndNode* is shown in Algorithm 4.5. Different with XOR structure, an AND structure contains no expression. Thus, there is no anomalies detected in AND split node in an SW. Lines 1 and 2 derive the LK and LW from the NLK and NLW of its parent node, respectively (Scheme 1). For each right branch of the AND

node, Lines 3~8 perform anomaly detection and collect the data flow by mutual recursive callings. Lines 9 and 10 calculate the NLK and NLW based on the flow data collected in Lines 3~8. For AND node in an SW, an element of NLK and NLW of AND split node should be removed if its artifact is operated in a branch of AND structure. At Line 9, *node->LK* ∩ *temp_node->LK* represents the element(s) of NLK of AND split node that exist in NLK of AND joint node in SW. This existence may indicate there is a branch of no operation for the artifact inside the element, which may make the calculation of NLK and NLW become wrong if the calculation only union all NLK and NLW from all branches, i.e., similar with calculation of NLK and NLW in XOR structure. A function *rmvIntersectOnHigherLevel* calculates *and_oa* of corresponding AND node since an AND structure may contain another AND structure. Lines 11~14 handle the mutual recursive routine by checking the existence of left_child. If the left_child exists, it handles left_child recursively. Otherwise, it replies the data flow used for analysis correspondingly.

Algorithm 4.5 ProcessAndNode
Input: ctree_node *node
Output: ctree_node *node *// return the data flow*
Global variables: SNA1, SNA2, SNA3, SNA4, and_oa

```
1  node->LK = node->parent->NLK;
2  node->LW = node->parent->NLW;
3  ctree_node *temp_branch;
4  ctree_node *branch = new ctree_node[node->right_
   branches.size];
5  for(int i=0;i < node->right_branches.size;i++){
6      branch[i] = SAD(right_branches[i]);
7      temp_branch = temp_branch ∪ branch[i];
8  }
9  node->NLK = removeFirstSet(temp_node->NLK,
           rmvIntersectOnHigherLevel(and_oa , (node->LK
           ∩ temp_node->NLK)));
10 node->NLW = removeFirstSet(temp_node->NLW,
           rmvIntersectOnHigherLevel(and_oa , (node->LK
           ∩ temp_node->NLW)));
11 if (node->left_child != null)
12    SAD(node->left_child);
13 else
14    return node;
```

2) Sequential anomaly in a loop structure

In an SW, a loop can be structured as a sequence of loop body X and loop control Y, where X may contain a sequence of operation(s) or a structural node, Y contains evaluation and/or initialization, and X may be executed after or before Y. Based on the structure, a loop can be divided into three types: (1) REPEAT, (2) FOR, and (3) WHILE, where a REPEAT loop is started with loop body, a FOR loop is started with initialization and evaluation, and a WHILE loop is started with evaluation. Besides, a loop body can be defined as one of two types: simple and complex, where a simple loop body is a structural node containing no loop, and a complex loop body is a structural node containing at least one loop. Since a sequential anomaly involve two operations on the same artifact, based on the structure of the loop, there are two sequential anomalies associated with a loop as follows:

Definition 4.2 Anomalies associated with a loop

The anomaly associated with a loop can be divided into two types:

1. A *pre-loop-end* anomaly is an anomaly whose 1st operation may exist either before or inside the loop and 2nd operation is inside the loop.
2. A *tailed-loop* anomaly is an anomaly whose 1st operation exists inside the loop and 2nd operation exists after the loop.

Pre-loop-end anomalies of REPEAT loop can be detected in sequence $X_1Y_1X_2Y_2$ and WHILE and FOR loops can be detected in sequence $Y_0X_1Y_1X_2Y_2$ [22]. For tailed-loop anomalies, the 1st operation of the anomalies in REPEAT loop can be found in $NLK(X_1Y_1)$ and $NLK(X_1Y_1)$ and WHILE and FOR loops can be found in $NLK(Y_0) \cap NLK(X_1Y_1)$ and $NLW(Y_0) \cap NLW(X_1Y_1)$ [22].

To detect pre-loop-end and tailed-loop anomalies in a C-tree, the switch statement in Algorithm 4.2 can be modified to handle LOOP node. In case of LOOP node, the detection routine *ProcessLoopNode* shown in algorithm 4.6 is executed. Algorithm 4.6 can simplify the anomaly detection by analyzing the first iteration of loop only, i.e., X_1Y_1 for REPEAT loop and $Y_0X_1Y_1$ for WHILE and FOR loops.

For pre-loop-end anomalies, Algorithm 4.6 detects the anomalies of REPEAT loop by analyzing X_1Y_1 and *loop_fo*, where *loop_fo* is used to store the information needed for the detection in X_2Y_2. Each element (*artifact_id*, *operation*, *node_id*) in *loop_fo* indicates the first operation of artifact (*artifact_id*) in the loop is *operation* in node *node_id*. Similar to *and_oa*, *loop_fo* is calculated in nodes of ACTIVITY node. Since X_2Y_2 is the same as X_1Y_1, the *loop_fo* can be derived in X_1Y_1 also. Besides, by analyzing X_1Y_1 and *loop_fo* in WHILE and FOR loops, it can detect the anomalies in $Y_0X_1Y_1$. For tailed-loop anomalies, Algorithm 4.6 stores the $NLK(Y_0)$ and $NLW(Y_0)$, and $NLK(X_1Y_1)$ and $NLW(X_1Y_1)$ separately. Thus, the NLKs and NLWs can be used to find the 1st operation of tailed-loop anomalies of all types of loop correspondingly.

In algorithm 4.6, Line 1 generates the SA sets of loop control Y. Lines 2 and 3 derive the LK and LW from NLK and NLW of its parent node, respectively (Scheme 1). This LK and LW contain the 1st operation(s) which exist before the loop. Lines 5~13 detect the anomalies and derive the NLK and NLW in Y_0, where Y_0 exists in loop_types FOR and WHILE. The temp_node at Line 4 is used to store the NLK and NLW of Y_0. Line 14 detects the anomalies and collects the flow data in Y_0X_1. Lines 15~18 detect the anomalies and collect the flow data in $Y_0X_1Y_1$. Lines 19~25 detect the anomalies and collect the flow data in $Y_0X_1Y_1X_2Y_2$. Lines 26~31 calculates the NLK and NLW of the loop. Lines 32~35 handle the mutual recursive routine by checking the existence of left_child. If the left_child exists, it handles left_child recursively. Otherwise, it replies the data flow used for analysis correspondingly.

Algorithm 4.6 ProcessLoopNode
Input: tree_node *node
Output: tree_node *node *//return the data flow*
Global variables: SNA1, SNA2, SNA3, SNA4, and_oa, loop_fo

```
1  processSAsets(node);
2  node->LK = node->parent->NLK;
3  node->LW = node->parent->NLW;
4  ctree_node *temp_node;
5  if(node->loop_type != "REPEAT"){
6     SNA1 += checkSeqAnomaly(node->LK, node->SAFR);
7     SNA2 += checkSeqAnomaly(node->LK, node->SAFK);
8     SNA3 += checkSeqAnomaly(node->LW, node->SAFK);
9     SNA4 += checkSeqAnomaly(node->LW, node->SAFW);
10
```

```
    node->NLK = removeFirstSet(node->LK, (node->SAFR ∪
11          node->SAFW ∪ node->SAFK)) ∪ node->SALK;
    node->NLW = removeFirstSet(node->LW, (node->SAFR ∪
12          node->SAFW ∪ node->SAFK)) ∪ node->SALW;
13    temp_node = node;
14  }
15  ctree_node *turn1 = SAD(loop_node-> right_branch);
16  SNA1 += checkSeqAnomaly(turn1->NLK, node->SAFR);
17  SNA2 += checkSeqAnomaly(turn1->NLK, node->SAFK);
18  SNA3 += checkSeqAnomaly(turn1->NLW, node->SAFK);
19  SNA4 += checkSeqAnomaly(turn1->NLW, node-> SAFW);
20  ctree_node * turn2;
    turn2->LK = removeFirstSet(turn1->NLK, (node-> SAFR ∪
21        node->SAFW ∪ node->SAFK)) ∪ node->SALK;
    turn2->LW = removeFirstSet(turn1->NLW, (node-> SAFR ∪
22        node->SAFW ∪ node->SAFK)) ∪ node->SALW;
23  SNA1 += checkSeqAnomaly(turn2->LK, getRead(loop_fo));
24  SNA2 += checkSeqAnomaly(turn2->LK, getKill(loop_fo);
25  SNA3 += checkSeqAnomaly(turn2->LW, getKill(loop_fo);
26  SNA4 += checkSeqAnomaly(turn2->LW, getRead(loop_fo);
27  node->NLK = turn1->NLK;
28  node->NLW = turn1->NLW;
29  if(node->loop_type != "REPEAT"){
30      node->NLK = node->NLK ∪ temp_node->NLK;
31      node->NLW = node->NLW ∪ temp_node->NLW;
32  }
33  if (node->left_child != null)
34     SAD(node->left_child);
35  else
       return node;
```

D. Example for Sequential Anomaly Detection

From the C-tree in Fig. 1, the recursive callings during anomaly detection based on Algorithm 4.2~4.6 are done by following the order below: (1) SAD(Start node), (2) SAD(Activity node A), (3) ProcessActivityNode(Activity node A), (4) SAD(Loop node), (5) ProcessLoop Node(Loop node), (6) SAD(AND node), (7) ProcessAndNode (AND node), (8) SAD(Activity node B), (9) ProcessActivityNode (Activity node B), (10) SAD(XOR node), (11) ProcessXor Node(XOR node), (12) SAD(Activity node C), (13) Process ActivityNode (Activity node C), (14) SAD(Activity node F), (15) ProcessActivityNode(Activity node F), (16) SAD(Activity node D), (17) ProcessActivityNode (Activity node D), (18) SAD(Activity node E), (19) ProcessActivity Node(Activity node E), and (20) SAD(End node).

Fig. 2 shows the order of above recursive callings, where the number next to the node represents the detection routine and the number on the arc represent the traverse routine. During the detection routine in a control node (i.e., XOR, AND, or LOOP nodes), it will handle the right branches firstly, and then handle the left child if it exists. For example, in the 5th recursion calling, it will handle the right branch by calling the 6th~17th recursive procedures firstly, after the procedures return the data flow, then handle the 18th procedure.

During the handling of right branch of LOOP node in the 5th recursion calling, i.e., the 6th~17th recursive callings: (1) In the 9th recursion calling, i.e., ProcessActivityNode(Activity node B), it detects the SNA4 anomalies: (V_2, A, B) and (V_3, A, B) by Line 7 in Algorithm 4.3. (2) In the 13th recursion calling, i.e., ProcessActivityNode(Activity node C), it detects the SNA4 anomalies: (V_4, A, C) and (V_5, A, C) by Line 7 in Algorithm 4.3, and an SNA3 anomaly (V_3, B, C) by Line 6 in Algorithm 4.3. (3) In the 17th recursion calling, i.e., ProcessActivityNode(Activity node D), it detects an SNA4 anomaly (V_3, A, D) by Line 7 in Algorithm 4.3.

After the handling of right branch of LOOP node (i.e., the 6th~17th recursive calling) in the 5th recursion calling, i.e., ProcessLoop Node(Loop node), it detects an SNA1 anomaly (V_1, C, B) by Line 22 in Algorithm 4.6, an SNA2 anomaly (V_1, C, C) by Line 23 in Algorithm 4.6, and the SNA4 anomalies: (V_2, B, B), (V_3, B, B), (V_3, B, D), (V_3, D, B), (V_3, D, D), (V_4, C, C), and (V_5, C, C) by Line 25 in Algorithm 4.6. In the 19th recursion calling, i.e., ProcessActivityNode (Activity node E), it detects the SNA2 anomalies: (V_2, C, E) and (V_3, C, E) by Line 5 in Algorithm 4.3, and the SNA3 anomalies: (V_4, C, E) and (V_5, C, E) by Line 6 in Algorithm 4.3.

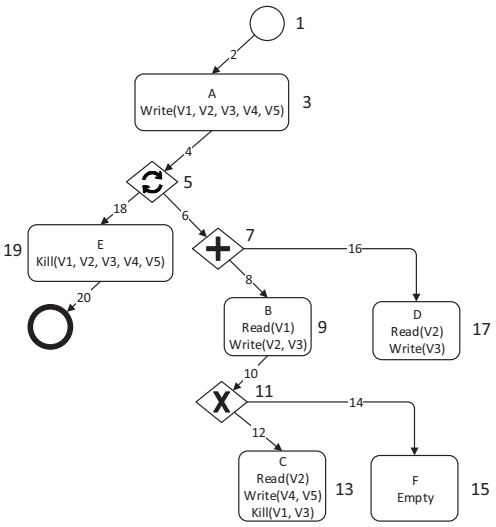

Fig 2. An example of recursive calling order based on Fig. 1

V. DISCUSSION

Table 1 shows the comparison of our approach with current approaches. The time complexity of our approach is split into two phases for discussion: the timing of traversing a C-Tree and the timing of data analysis inside each node of a C-Tree.

Let S_{AND}, S_{XOR}, S_{LOOP} represent the number of AND, XOR, LOOP split nodes in an SW correspondingly, and N_{Act} represent the number of Activity nodes in an SW. The number of nodes in an SW, $N_{workflow}$, is $(S_{AND}+S_{XOR}+S_{LOOP})*2+N_{Act}+2$, where the rightmost 2 represents the START node and END node. The number of nodes in the corresponding C-Tree, named as NC-Tree, is $S_{AND}+S_{XOR}+S_{LOOP}+N_{Act}+2$, because the joint nodes of control structure in the SW are removed. Clearly, $N_{workflow}$ is bigger than $N_{C\text{-}tree}$ and the time complexity of transformation is O(n).

For the traversing of a C-Tree, the algorithms initiate the recursion with the left_child of START node. Totally, each node in a C-Tree is traversed once at most. The time complexity of traversing C-Tree is O(n).

The data analysis inside each node of a C-Tree is under a sequence of set operations, where each element in a set represents an artifact. The set operators adopted in our analysis are $\cup$, $\cap$, $\setminus$ and the time complexities are the same. Let each expression in each algorithm have no more than K operands. If we adopt Hash table to access data set, the time complexity of each operator is constant. Thus, the time

complexity of each expression is constant. As a result, the time complexity of data analysis is constant O(n).

For the space complexity, since our approach can analyze the loop in a single iteration only, it will reduce the space to store i iteration of the loop. Suppose the anomaly detection for each node in a C-tree needs n unit of space, in worst case, i.e., all nodes in the C-tree is a loop, the space complexity is (S) = n.

Table 1: the comparison of our approach with current approaches.

	Wang et al. [10]	Wang et al. [21]	Sun et al. [11]	Our method
Time complexity	$O(n^2)$	O(n)	$O(n^3)$	O(n)
Space complexity	(S) = 2n	(S) = 2n	$(S) = n^2$	(S) = n

REFERENCES

[1] Workflow Management Coalition, Workflow Management Coalition: Terminology & Glossary, Document Number WFMC-TC-1011, 1999

[2] N. R. Adam, V. Atluri, W.-K., Huang. "Modeling and analysis of workflows using Petri nets". Journal of Intelligent Information Systems, 10.2: 131-158, 1998.

[3] W. M. P. van der Aalst. Three good reasons for using a Petri-net-based workflow management system. in Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96), p. 179-201, 1996.

[4] W. M. P. van der Aalst. The application of Petri nets to workflow management. Journal of circuits, systems, and computers, 8.01: 21-66, 1998.

[5] W. M. P. van der Aalst, and A. H. M. ter Hofstede, "Verification of workflow task structures: A petri-net-based approach," in Information systems, Vol. 25, issue 1, pp. 43-69, 2000.

[6] W. M. P. van der Aalst, T. Bastenet, H. M. W. Verbeek, and P.A.C. Verkoulen, "Adaptive workflow: an approach based on inheritance," in International Joint Conference on Artificial Intelligence, pp. 36-45, 1999.

[7] H. J. Hsh and F. J. Wang, "An incremental analysis for resource conflicts to workflow specifications," in Journal of Systems and Software, Vol. 81, issue 10, pp. 1770-1783, 2008.

[8] H. J. Hsh and F. J. Wang, "Using Artifact Flow Diagrams to Model Artifact Usage Anomalies," in Computer Software and Applications Conference, Vol. 2, pp. 275-280, 2009.

[9] H. J. Hsh and F. J. Wang, "Detecting artifact anomalies in temporal structured workflow as reusable assets," in the Proceedings of 35th Annual IEEE International Computer Software and Applications Conference Workshops, pp. 362-367, 2011.

[10] C. H. Wang and F. J. Wang, "Detecting artifact anomalies in business process specifications with a formal model," in the Journal of Systems and Software, ol. 82, pp. 1600-1619, 2009.

[11] S. X. Sun, J. L. Zhao, J. F. Nunamaker, and O. R. L. Sheng, " Formulating the Data-Flow Perspective for Business Process Management," in Information Systems Research, Vol. 17, issue 4, pp. 374-391, 2006.

[12] S. Sadiq, M. Orlowska, and W. Sadiq, "Data flow and validation in workflow modeling," in Conferences in Research and Practice in Information Technology, Vol. 27, pp. 207-214, 2004.

[13] C. L. Hsu and F. J. Wang, "Analysing inaccurate artifact usages in workflow specifications," in IET Software, Vol. 1, issue 5, pp. 188-205, 2007.

[14] B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler, "On Structured Workflow Modelling", Lecture Notes in Computer Science, Vol. 1789, 2000, pp. 431-445.

[15] C. Karamanolis, D. Giannakopoulou, J. Magee, and S. M. Wheater, "Model checking of workflow schemas," in the Proceedings of IEEE Fourth International Enterprise Distributed Objects Computing Conference, pp. 170-179, 2000.

[16] W. M. P. van der Aalst and T. Basten, "Inheritance of workflows: an approach to tackling problems related to change," in the Journal of Theoretical Computer Science, Vol. 270, pp.125-203, 2002.

[17] F. J. Wang and Mandalapu, Parameswaramma, "Analyzing Artifact Anomalies in a Temporal Structural Workflow for SBS," in the Proceedings of Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International, pp. 480-485, 2014.

[18] W. Sadiq and M. E. Orlowska, "Analyzing process models using graph reduction techniques," in the Journal of Information systems, pp. 117-134, 2000.

[19] W. M. P. van der Aalst, K. M. van Hee, et al, "Soundness of workflow nets: classification, decidability, and analysis," in the Journal of Formal Aspects of Computing, pp. 333-363, 2011.

[20] H. Li, Y. Yang, and T. Y. Chen, "Resource constraints analysis of workflow specifications," in the Journal of Systems and Software, pp. 271-285, 2004.

[21] F. J. Wang, A. Chang, and T. Lu, "Improving Workflow Anomaly Detection with a C-Tree," in the Proceedings of 41th Annual IEEE Computer Software and Applications Conference, Vol. 2, pp. 437-444, 2017.

[22] F. Fahmi, P.-S. Huang, and F.-J. Wang, "Improving the Detection of Artifact Anomalies in a Structured C-tree Program," to be Submitted.