These algorithms perform user interface derivation for business processes. They are developed by Lei Han, Jian Yang, and Weiliang Zhao in Lei Han’s PhD Project. These algorithms are related with the described work in Lei Han’s PhD thesis, paper in ICSOC 2016 with title “User Interface Derivation Based on Role-Enriched Business Process Model”, and under-reviewed paper for the journal IEEE Transactions on Knowledge and Data Engineering with title “User Interface Derivation for Business Processes”.

1 Algorithms for Task Abstraction and Aggregation

This section introduces the algorithms for task abstraction and aggregation, which calls the elementary operations. In the algorithms, the tasks related to a particular user role will be reserved in the output abstracted and aggregated BP, while the tasks irrelevant to this user role will be abstracted and aggregated into abstracted nodes in the output abstracted and aggregated BP. In order to realize this goal, we proceed recursively to detect and handle the elements on each granularity level of the role-enriched BP. In this procedure, the recursive function $BP_{AbsAgg}$ in Algorithm 3 is the working horse of this algorithm. The control flow patterns at non-finest granularity level of the role-enriched BP belong to Complex BP Fragments and they are abstracted/aggregated by the function $AbsComplexFrag$. The control flow patterns at the finest granularity level of the role-enriched BP belong to Basic BP Fragments and they are abstracted/aggregated by applying the function $AbsBasicFrag$.

A Basic BP Fragment in a role-enriched BP must satisfy one of the following conditions:

- if the control flow pattern of the fragment is Strict-order Sequential or Free-order Sequential, then all the elements of this fragment must only be individual tasks. (For example in Figure ??, (a) and (b) are Basic BP Fragments, while (g) is a Complex BP Fragment.)
- if the control flow pattern of the fragment is Parallel-A, Parallel-B, or Parallel-C, then the control flow pattern of each branch of this fragment must only be Strict-order Sequential or Free-order Sequential. (c) are Basic BP Fragments, while (h) is a Complex BP Fragment.)
- if the control flow pattern of the fragment is conditional, then
  - the control flow pattern of each branch of this fragment must only be Strict-order Sequential or Free-order Sequential and
In order to abstract and aggregate the tasks in a complex BP fragment, the first step is to identify the control flow pattern $CFPattern$ and the related elements in $TaskSet$ on the coarsest granularity level of the input fragment $CFrag$ (line 2). We use different methods to deal with different elements in $TaskSet$ according to the types of the elements.

We assume that the abstraction and aggregation is for user role $r_1$. In $TaskSet$, there are four categories of elements: (1) a task $Task_{Local}$
that is executed by $r_1$, (2) a task $Task_{Foreign}$ that is not executed by $r_1$, (3) a task $Task_{Multi-Or}$ that can be executed by $r_1$ or other user roles, (4) a task $Task_{Multi-And}$ that must be executed by both $r_1$ and other user roles, and (5) a block of tasks with one entry gateway and one exit gateway, inside this block there exist complex control flow relations between/among these tasks.

Line 3 represents $TaskSet$ does not contain $Task_{Multi-Or}$ or $Task_{Multi-And}$. In this case, if $CFPattern$ matches one of the elementary operations, the input role-enriched BP fragment is transformed by this matched elementary operation (line 4 and line 5); if $CFPattern$ cannot match any of the elementary operations, each $Task_{Foreign}$ is abstracted by Single-Abs-Agg-1 (line 6 and line 7). After that, the transformed fragment is examined to see if it has adjacent abstracted nodes or not. If there exist adjacent abstracted nodes, they are aggregated as single abstracted nodes (line 8). The other situation is $CFEleSet$ that contains $Task_{Multi-Or}$ or $Task_{Multi-And}$ (line 9), $Task_{Foreign}$ are abstracted by Single-Abs-Agg-1, and $Task_{Multi-Or}$ and/or $Task_{Multi-And}$ are abstracted by Single-Abs-Agg-2 and/or Single-Abs-Agg-3 (line 10). Then we examine the transformed fragment for adjacent abstracted nodes. If there exist adjacent abstracted nodes, they are aggregated as single abstracted nodes (line 11). Lastly, the transformed result is returned (line 12).

**Handling Basic BP Fragments** We use function AbsBasicFrag to deal with the Basic BP Fragments of role-enriched BP. The input is a Basic BP Fragment, and the output is the transformed Basic BP Fragment.

In order to abstract and aggregate the tasks in a Basic BP Fragment $BFragment$, the first step is to identify the control flow pattern $cfPattern$ and the related elements in $cfEleSet$ of $BFragment$ (line 2). We use different methods to the deal with the $BFragment$ according to different $cfPattern$.

If $cfPattern$ is **Strict-order Sequential**, **Free-order Sequential**, **Strict-loop**, or **Free-loop** (line 3), we check whether $BFragment$ contains $MultiRoleTask$. If no $MultiRoleTask$ is included, $BFragment$ is transformed by using corresponding elementary operations (line 4 and line 5); if there exist $MultiRoleTasks$ in $BFragment$, each task in $cfEleSet$ of $BFragment$ is transformed by using Single-Abs-Agg-1 or Single-Abs-Agg-2, and $BFragment'$ is the result. Then $BFragment'$ is transformed using TransformByTree as $BFragment''$. After that, we check the adjacent abstracted nodes in $BFragment''$ and aggregate them as single abstracted nodes (from line 6 to line 9).

If $cfPattern$ is **Parallel-A**, **Parallel-B**, **Parallel-C**, or **Conditional** (line 10), we check again whether $BFragment$ contains $MultiRoleTask$.
Algorithm 2: Function for Handling Basic BP Fragment

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Function AbsBasicFrag(BasicFrag BFrag)</td>
</tr>
<tr>
<td>2</td>
<td>identify cfPattern and cfEleSet of BFrag;</td>
</tr>
<tr>
<td>3</td>
<td>if cfPattern = StrictSeq, FreeSeq, StrictLoop, or FreeLoop then</td>
</tr>
<tr>
<td>4</td>
<td>if cfEleSet does not contain MultiRoleTask then</td>
</tr>
<tr>
<td>5</td>
<td>BFrag&quot; = transform (BFrag, Sequential – Abs – Agg – 1/ – 2) or Loop – Abs – Agg – 1/ – 2);</td>
</tr>
<tr>
<td>6</td>
<td>else if cfEleSet contains MultiRoleTask then</td>
</tr>
<tr>
<td>7</td>
<td>BFrag' = transform (BFrag, Single – Abs – Agg – 1/ – 2);</td>
</tr>
<tr>
<td>8</td>
<td>BFrag&quot; = TransformWithTree (BFrag‘);</td>
</tr>
<tr>
<td>9</td>
<td>examine and combine adjacent abstracted nodes in BFrag&quot;;</td>
</tr>
<tr>
<td>10</td>
<td>else if cfPattern = ParallelA, ParallelB, ParallelC, or Conditional then</td>
</tr>
<tr>
<td>11</td>
<td>if cfPattern matches elementaryOperation then</td>
</tr>
<tr>
<td>12</td>
<td>BFrag&quot; = transform (BFrag, elementaryOperation);</td>
</tr>
<tr>
<td>13</td>
<td>else</td>
</tr>
<tr>
<td>14</td>
<td>foreach branch ∈ BFrag do</td>
</tr>
<tr>
<td>15</td>
<td>branch' = transform (branch, Single – Abs – Agg – 1/ – 2);</td>
</tr>
<tr>
<td>16</td>
<td>BFrag‘ is the result;</td>
</tr>
<tr>
<td>17</td>
<td>examine and combine adjacent abstracted nodes in BFrag&quot;;</td>
</tr>
<tr>
<td>18</td>
<td>else if cfEleSet contains MultiRoleTask then</td>
</tr>
<tr>
<td>19</td>
<td>foreach branch ∈ BFrag do</td>
</tr>
<tr>
<td>20</td>
<td>branch' = transform (branch, Single – Abs – Agg – 1/ – 2);</td>
</tr>
<tr>
<td>21</td>
<td>BFrag‘ is the result;</td>
</tr>
<tr>
<td>22</td>
<td>BFrag&quot; = TransformWithTree (BFrag‘);</td>
</tr>
<tr>
<td>23</td>
<td>examine and combine adjacent abstracted nodes in BFrag&quot;;</td>
</tr>
<tr>
<td>24</td>
<td>return the transformed fragment BFrag&quot;;</td>
</tr>
</tbody>
</table>
When no MultiRoleTask is included (line 11), we check if $B\text{Frag}$ can be transformed directly by using elementary operations. If suitable elementaryOperation can apply, $B\text{Frag}$ is transformed accordingly as $B\text{Frag}''$ (line 12 and line 13); if no suitable elementary operation can apply, we only abstract single tasks on each branch of $B\text{Frag}$ using Single-Abs-Agg-1 or Single-Abs-Agg-2 and $B\text{Frag}''$ is the result. Then we check the adjacent abstracted nodes on each branch of $B\text{Frag}''$ and aggregate them as single abstracted nodes (from line 14 to line 18), the result is named as $B\text{Frag}''$.

When MultiRoleTask(s) is/are included, single tasks on each branch of $B\text{Frag}$ is transformed by using Single-Abs-Agg-1 or Single-Abs-Agg-2 and $B\text{Frag}'$ is the result. Then $B\text{Frag}'$ is transformed by using TransformByTree and $B\text{Frag}''$ is returned. After that, we check the adjacent abstracted nodes in $B\text{Frag}''$ and aggregate them into single abstracted nodes (from line 19 to line 24).

Finally, the fragment $B\text{Frag}''$ is returned as the result (line 25).

In the above algorithm, the function TransformByTree is used to transform a Sequential fragment containing the Task-Abs Blocks to a Conditional fragment using tree graph. The input of this function is a control flow pattern Strict-order Sequential or Free-order Sequential, where the element set contains: (1) at least one Task-Abs Block, (2) zero to many tasks performed by $r_1$, and (3) zero to many abstracted nodes. The output of this function is a Conditional fragment.

The first step is to generate a tree diagram for the input block. In this step, single tasks and abstracted nodes are added onto the tree graph ($Task_B$ and $ABS_{DE}$); each Task-Abs Block is separated as individual branches on the tree graph ($Task_{A-ABS_{DE}}$ block and $Task_{C-ABS_C}$ block). The second step is to transform the tree graph to a Conditional fragment. In this step, each path in the tree graph is transformed as a branch of the Conditional fragment.

**Abstracting and Aggregating Tasks in Role-enriched BP** In this section, we introduce Algorithm 3 for task abstraction and aggregation. The input is a role-enriched BP, and the output is an abstracted and aggregated BP. This algorithm is realized by using a recursive function BPAbsAgg, which iteratively calls itself. In the recursive way, each granularity level of the role-enriched BP is reached, ranging from coarsest granularity level (the role-enriched BP itself) to the finest granularity level (the Basic BP Fragments). In doing so, tasks on each granularity level are dealt with by using functions AbsComplexFrag and AbsBasicFrag.
Algorithm 3: Task Abstraction and Aggregation

**Input**: RoleEnrichedBP

**Output**: AABP

1. AABP = ∅;
2. BPAbsAgg(RoleEnrichedBP);
3. return AABP;
4. Function BP_absAgg(RoleEnrichedBP RoleEnBP)
5. RoleEnBP\(^\prime\) = AbsComplexFrag(RoleEnBP);
6. if AABP == ∅ then
   7. add RoleEnBP\(^\prime\) to AABP;
else
7. update RoleEnBP in AABP with RoleEnBP\(^\prime\);
8. identify cfElementSet of RoleEnBP\(^\prime\);
9. foreach element ∈ cfElementSet do
10.   if element is BasicBPFragment then
11.      element\(^\prime\) = AbsBasicFrag(element);
12.      update element in AABP with element\(^\prime\);
else
13.      BP_absAgg(element);

When a role-enriched BP RoleEnrichedBP is input into this recursive function, function AbsComplexFrag is used to deal with the elements on the coarsest granularity level of RoleEnrichedBP (line 4). In case the input RoleEnrichedBP has only one granularity level, where the elements on this level equal to those on the finest level, the function AbsComplexFrag deals with this RoleEnrichedBP in the same way as the function AbsBasicFrag does. After handled by AbsComplexFrag, the handled RoleEnrichedBP\(^\prime\) is copied onto the initialized AABP (line 6 and line 7). Then the element set cfElementSet on the coarsest level of RoleEnrichedBP\(^\prime\) is identified (line 10) to deal with the tasks on a finer level \(L_1\). Till this step, two possible results exist: (1) if \(L_1\) is the finest level, the function AbsComplexFrag is used to deal with the tasks on this level \(L_1\), and the dealt result replaces the counter parts in AABP; (2) if \(L_1\) is not the finest level, the RoleEnrichedBP\(^\prime\) is re-input into the recursive function BP_absAgg to handle the finer but non-finest level (line 15 and line 16). From the second iteration, the result of the function AbsComplexFrag in this iteration continuously replace the result in AABP of last iteration. The recursive function BP_absAgg finalizes after all the tasks on the finest level are handled. Then the final abstracted and aggregated BP AABP is derived and returned (line 3).
2 Algorithm for Data Relationship Extraction

In order to extract the data relationships from an AABP, the control flow patterns of the AABP and the data operation patterns inside each individual task are identified and elementary operations are applied accordingly. A tree graph representing the extracted data relationships is built up by using Algorithm 4. This algorithm comprises two functions DeriveDataRelationships that deals with the control flow relations between tasks and abstracted nodes in a AABP, and HandleDataOperationFlowInTask that deals with the data operation flow inside tasks of the AABP.

The function DeriveDataRelationships is utilized to extract the data relationships from the an AABP. The input is an AABP for user role \( r \), and the output is a tree fragment set that specifies the data relationships extracted from the input AABP.

The data relationship extraction of the input AABP is realized in a recursive way. When an abstracted and aggregated business process AABP is input (line 1), the control flow pattern \( cfPattern \) on the coarsest granularity level of AABP and the related elements \( cfElementSet \) constituting the \( cfPattern \) are both identified (line 3). A tree graph \( TreeGraph \) is initialized (line 19). If an identified element \( cfElement \) in \( cfElementSet \) is task (line 5), we find the data operation flow \( df_{cfElement} \) from this task \( cfElement \) (line 6). The function TransformDataFlowInTask is used to extract the tree graph fragment set from the \( df_{cfElement} \) (line 7), and this set is added to the initialized \( TreeGraph \) (line 8).

When \( cfElementSet \) does not contain any complex control flow structure, we transform the control flow on the coarsest granularity level of the AABP into a tree graph fragment. The first step is to find suitable elementary operation according to the identified \( cfPattern \). This found elementary operation extracts the data relationships \( treeFragment \) from \( cfElementSet \) (line 11). The second step is to assign a structure reference to the the extracted \( treeFragment \): if the \( cfPattern \) is on the coarsest granularity level of AABP, the \( ProcessStructRef \) is assigned to the \( treeFragment \) (line 12 and line 13); otherwise, a \( NormalStructRef \) is assigned to the \( treeFragment \) (line 14 and line 15). The third step is to add the \( treeFragment \) to the \( TreeGraph \) (line 16). Till now, the data relationship extraction of the coarsest granularity level of AABP is completed. If an identified element \( cfElement \) in \( cfElementSet \) is a complex control flow structure (line 9), we input this \( cfElement \) into the function DeriveDataRelationships for recursive processing. The recursively processed result \( cfElement \) from the function TransformDataFlowInTask
Algorithm 4: Data Relationship Extraction

**Input**: an AABP for user role $r$

**Output**: a TreeGraph

1. **Function DeriveDataRelationships**(AbsAggBusinessProcess $aabp$)
   
   2. $TreeGraph = \emptyset$;
   3. identify $cfPattern$ and $cfElementSet$ at coarsest granularity level of $aabp$;
   4. **foreach** $cfElement$ in $cfElementSet$ **do**
      
      5. **if** $cfElement$ is Task **then**
         
         6. get data operation flow $df_{cfElement}$ from $cfElement$;
         7. result = **HandleDataOperationFlowInTask**(df$_{cfElement}$);
         8. add result to $TreeGraph$;
      
      9. **else if** $cfElement$ is ComplexStructure **then**
         
         10. $cfElement = \text{DeriveDataRelationships}(cfElement)$;
         
      11. transform $cfElementSet$ to $treeFragment$ using elementary operation according to $cfPattern$;
      12. **if** $cfPattern$ is on the coarsest granularity level **then**
         
         13. assign ProcessStructRef to $treeFragment$;
      14. **else**
         
         15. assign NormalStructRef to $treeFragment$;
      16. add $treeFragment$ to $TreeGraph$;
      17. **return** $TreeGraph$;

18. **Function HandleDataOperationFlowInTask**(DataOperationFlow $df$)
   
   19. $TreeFragmentSet = \emptyset$;
   20. identify $dfPattern$ and $dfElementSet$ at coarsest granularity level of $df$;
   21. **foreach** $dfElement$ in $dfElementSet$ **do**
      
      22. **if** $dfElement$ is not DataItem **then**
         
         23. $dfElement = \text{HandleDataOperationFlowInTask}(dfElement)$;
      
      24. transform $dfElementSet$ to $treeFragment'$ using elementary operation according to $dfPattern$;
      25. assign NormalStructRef to $treeFragment'$;
      26. add $treeFragment'$ to $TreeFragmentSet$;
      27. **return** $TreeFragmentSet$;
replaces the input \textit{cfElement} (line 24). After all the complex structure are handled, the extracted \textit{TreeGraph} is returned as the output of \texttt{DeriveDataRelationships} (line 17).

The function \texttt{TransformDataFlowInTask} is adopted to extract the data relationships from the data operation flow inside a task of the AABP. The input is a data operation flow that specifies the execution flows of data items operated inside a tasks of the AABP, and the output is a tree fragment set that specifies the extracted data relationships.

The data relationship extraction inside a task of the AABP is realized in a recursive way. When a data operation flow \textit{df} is input (line 18), the data operation pattern \textit{dfPattern} on the coarsest granularity level of \textit{df} and the related elements \textit{dfElementSet} constituting the \textit{dfPattern} are both identified (line 20). A set of tree graph fragments \textit{TreeFragmentSet} are also initialized (line 19). If each of identified element in \textit{dfElementSet} is data item (line 21), we transform the data operation flow on the coarsest granularity level into a tree graph fragment. The first step is to find suitable elementary operation according to the identified \textit{dfPattern}. This found elementary operation extracts the data relationships \textit{treeFragment} from \textit{dfElementSet} (line 24). The second step is to assign \textit{NormalStructRef} to the extracted \textit{treeFragment}. The third step is to add the \textit{treeFragment} to the initialized \textit{TreeFragmentSet} (line 26). Till now, the data relationship extraction of the coarsest granularity level of \textit{df} is completed. If an identified element \textit{dfElement} in \textit{dfElementSet} is a complex data operation structure (line 21 and line 22), we input this \textit{dfElement} into the function \texttt{TransformDataFlowInTask} for recursive processing. The recursively processed result \textit{dfElement} from the function \texttt{TransformDataFlowInTask} replaces the input \textit{dfElement} (line 23). After all the complex structure are handled, the extracted \textit{TreeFragmentSet} is returned as the output of \texttt{TransformDataFlowInTask} (line 27).

3 Algorithm for UI Derivation

In this section, we introduce the algorithm for UI derivation, which operate the rules. All these rules are categorized as two groups (\textit{Group 1} and \textit{Group 2}).


Rules in Group 2 are about the separation of inside and outside data of a data operation pattern. This group includes Parallel-A-Constraint-3, Conditional-Constraint-2, Loop-Constraint-2, Loop-Constraint-4, Parallel-A-Recommendation-2.

<table>
<thead>
<tr>
<th>Algorithm 5: UI Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: TreeGraph</td>
</tr>
<tr>
<td><strong>Output</strong>: UIFlow</td>
</tr>
<tr>
<td>1. retrieve headTreeFragment with ProcessStructRef from TreeGraph;</td>
</tr>
<tr>
<td>2. generate UIFlowFrag from headTreeFragment by applying rules in Group 1;</td>
</tr>
<tr>
<td>3. set UIFlow = UIFlowFrag;</td>
</tr>
<tr>
<td>4. UIDerivation(headTreeFragment);</td>
</tr>
<tr>
<td>5. process UIFlow by applying rules in Group 2;</td>
</tr>
<tr>
<td>6. return UIFlow;</td>
</tr>
</tbody>
</table>

7. Function UIDerivation(TreeFragment treeFrag)
   8. get structRef from treeFrag;
   9. if structRef is not ProcessStructRef then
      10. generate UIFlowFrag from treeFrag by applying rules in Group 1;
      11. update the container holding structRef with UIFlowFrag;
   12. foreach node of treeFrag do
      13. if node is CDENode then
          14. get the value structRef′ from node;
          15. retrieve treeFrag′ with structRef′ from TreeGraph;
          16. UIDerivation(treeFrag');

The Algorithm 5 is built up to derive the UI from a tree graph. The input is a tree graph set that specifies the data relationships for a particular user role; and the output is a the user interface flow which specifies the derived BP UI for the user role. The function UIDerivation is the working horse of this algorithm. When a tree graph TreeGraph is input into the algorithm, the first step is to retrieve the the tree fragment headTreeFragment, which has ProcessStructRef from TreeGraph (line 1). According to the data relationship pattern of headTreeFragment, suitable rule from Group 1 is found to derive the UI flow fragment UIFlowFrag from headTreeFragment (line 2). This UIFlowFrag is copied to the initialized UI Flow UIFlow (line 3). Then headTreeFragment is input into the function UIDerivation (line 4). In this function, the structRef of the input tree fragment is found for judgement (line 8).
If `structRef` is not `ProcessStructRef` (line 9), the input tree fragment must not be the `headTreeFragment`. Under this condition, `UIFlowFrag` is derived from the input tree fragment according to suitable rule in `Group 1` (line 10). Then the the `container` holding `structRef` in `UIFlow` is replaced with `UIFlowFrag` (line 11). Next, the nodes inside the input tree fragment is examined. If there exist any CDE node `CDENode` (line 13), the tree graph fragment `treeFrag'` pointed by this `CDENode` is found (line 14 and 15). This `treeFrag'` is re-input into `UIDerivation` for recursive processing (line 16). After all the tree fragments in the `TreeGraph` are handled, the rules in `Group 2` are used in `UIFlow` for secondary separation (line 5). Then the final result `UIFlow` is returned as the output of the algorithm (line 6).