

IBM Research Report

Business Process Modeling in Abstract Logic Tree

Ying Liu, Jian Wang, Jun Zhu, Haiqi Liang, Zhong Tian, Wei Sun

²IBM Research Division

China Research Laboratory

HaoHai Building, No. 7, 5th Street

ShangDi, Beijing 100085

China



Research Division

Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Business Process Modeling in Abstract Logic Tree

Ying Liu, Jian Wang, Jun Zhu, Haiqi Liang, Zhong Tian, Wei Sun
IBM China Research Lab
4F, Haohai Building, 5th Shangdi Street, Haidian District, Beijing China 100085
{aliceliu, wangwj, zhujun, lianghq, tianz, weisun}@cn.ibm.com

ABSTRACT

Business process models are usually defined in a graphical modeling language. Most business process modeling languages are the analog of flow chart and UML Activity Diagram, which allows unstructured flow structures. Unstructured process models make it difficult to transform it to a structured business process model, such as BPEL4WS. This paper proposes to represent the structure of a business process model with a special tree structure, Abstract Logic Tree. The concept and approach of Abstract Syntax Tree of programming language field is suggested to be applied to business process modeling field in this paper. Several graph transformation rules are developed for the transformation from an unstructured process model to an ALT. Detecting unstructured loops is the critical point for the transformation. DJ Graph is used to detect unstructured loops in this paper. The equivalence between a process model and its ALT is proven. The efforts in the paper make the analysis and manipulation against process models can be easily done on tree-based internal representation. ALT can be regarded as a foundation for parsing the structure and analyzing structure properties of business process models.

Categories and Subject Descriptors

D.3.1 [Programming Languages]: Formal Definitions and Theory –*Semantics, Syntax.*

General Terms

Languages, Theory.

Keywords

Business Process Model, Abstract Logic Tree, DJ Graph, Structure Equivalence

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

1. INTRODUCTION

1.1 Motivation

Business process models can be used not only as the specification of business behaviors for understanding, but also for the automation of business operation. Business process modeling is becoming more and more popular in both business and technologies. Some business process modeling languages are designed for business level requirements capturing. Most of them are the analog of flow charts [6] and UML Activity Diagram [7]. These languages are often used by business people, so they are often designed to be flexible with little constraints, for example, unstructured flow structures are allowed in these languages. That is to say, the arbitrary “Goto” control flow and unstructured loops are allowed. Some process modeling languages are designed for the business process management and automation at IT level, such as BPML [1] and BPEL4WS[2]. They often follow the structured modeling principle leveraged from the structured programming, in which only structured blocks are allowed. While creating a business integration solution basing on Model Driven Architecture (MDA)[3] or transforming a business level model to an IT level model, how to deal with an unstructured flow is a big challenge.

In different scenarios, business process models need to be analyzed, optimized, translated and checked on the correctness of syntax and semantics, and so on. It is necessary to develop a formal representation for business process models to enable the easy manipulation, just like the Abstract Syntax Tree (AST) [4,5], which is generally used as internal representation of a programming language for its syntax checker, translator, optimization processor, interpreter, or compiler. If the essential structure of a BPM is formally represented, then the deep analysis to it can be done based on the essential structure, which can help to reduce the complexity of structure analysis.

In this paper, we proposed to use a special tree structure as the internal representation for business process modeling languages. This special tree structure is Abstract Logic Tree (ALT). The transformation approach and their equivalence proof is given in this paper. The value of this effort includes two aspects:

(1) The advantage of representing BPM with ALT is that the structure of tree is isomorphic, which makes it easy to design a simple environment to analyze and manipulate business process via doing it on the tree representation.

(2) The transformation from unstructured flow to structured tree provides a foundation and reference for transforming unstructured business processes to structured ones.

In order to represent the structure of BPM with ALT, a set of transformation rules are provided. Moreover, we proved that the structure of a BPM is equivalent to the corresponding ALT that can be obtained by using the set of transformation rules. The loop structure may result in a very complex BPM. How to identify all the loops of a BPM is also an important research topic. This paper extends the algorithm of identifying irreducible loops using DJ graphs [8,9] to support identifying nest irreducible loops.

1.2 Business Process Model

Most business process modeling languages are the variations and extensions of flow charts and UML Activity Diagram. Here, we give some basic elements in business process modeling languages in Figure 1. They are enough to model complex process logics and to demonstrate our ideas – representing the structure of business process models using ALT.

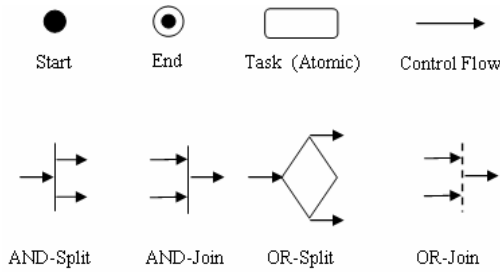


Figure 1. Some Elements of BPM

The Start and End element represent the beginning and end of a business process separately. A Task is an atomic activity in a process. A Control Flow is used to show the order that activities will be performed in a process. An AND-Split is used to specify that two or more activities can be performed concurrently, rather than sequentially. The element “AND-Join” is used to combine two or more parallel paths into one path. It is possible that different parallel paths will arrive to the AND-Join point at different time. However, only all the parallel paths arrive to this point, can different paths be combined to one path. The OR-Split represents that only one of a set of alternatives may be chosen in runtime. The choice is based on the computation results of conditions. The element “OR-Join” is used to combine at least one path into one path. Different from “AND-Join”, it is not required all the paths arrive to this point. Once one path arrives to the point, it will continue the path of the following.

Because business process modeling language has little constraints in syntax structure, the above elements may result in arbitrary complex business process models. An example is given in figure 2. A business process is a set of logically related business activities that combine to deliver something of value to a customer [10]. It is a composition of activities and it directs the execution of these activities. No matter what are drivers behind a business process design, it is necessary to understand the execution logic of its activities. The execution logic can also be called execution order of activities. An execution order of a process model is called a process path. From Start to End of a process model, there are several process paths. We use a sequence of activity labels to represent a process path. For example, $p = \langle a; b; d; f; g; i; n; q \rangle$ is a path of the business process defined in Figure 2.

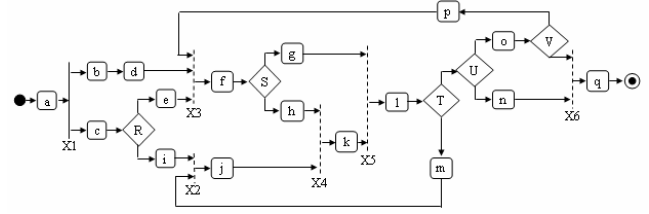


Figure 2. Example of BPM

Although this example doesn't look very complex, it is not easy to identify all the paths. For example, complex loops in this model are entangled so that it is difficult to detect what activities should be included in each loop. Furthermore, different people may find out different loops from their own perspectives. One person may find out it includes two loops, $(x2, j, x4, k, x5, l, T, m)$ and $(x3, f, S, h, g, x4, k, x5, l, T, U, o, V, p)$. But another person may believe it includes three loops, $(x3, f, S, h, g, x4, k, x5, l, T, U, o, V, p, x2, j, m)$, $(x2, j, x4, k, x5, l, T, m)$, and $(x3, f, S, h, g, x4, k, x5, l, T, U, o, V, p)$. Therefore, it is necessary to provide an algorithm to detect all the loops of a BPM, which will be introduced in section 4. On the other hand, the business process diagram is unstructured and entangled. In order to clearly represent the structure of a BPM, how to capture the essential structure of it using ALT will be introduced in section 3.

Using ALT to represent the structure of a business process model omits some syntax details and some concrete data information, however, the paths of activities in the business process model should be captured completely in the ALT. If two business process models define the same set of paths, these two models are called structure equivalent. The following is the definition of structure equivalent of business process models:

DEFINITION 1: Structure Equivalent of business process models

Two business process models $F1$ and $F2$ are said to be structure

equivalent, in short $F1 \stackrel{S}{=} F2$, if for every path $p = \langle t1; \dots; tn \rangle$ of $F1$ there is a path $q = \langle t1'; \dots; tn' \rangle$ of $F2$ such that

$$\forall 1 \leq i \leq n: ti = ti'$$

and vice versa.

From the definition, it is easy to prove that $\stackrel{S}{=}$ is an equivalent relation. If a business process model is represented with an ALT, it is necessary to prove the business process model is structure equivalent with the ALT.

The remainder of this paper is structured as the following. Section 2 briefly introduces abstract logic tree, including its syntax and informal semantics. Section 3 discusses how to analyze the structure of a business process model based on ALT, where identifying the loops of a business process using DJ graph will be studied in detail. In section 4, we introduce how to transform a business process model into a structure equivalent ALT by using transformation rules. The conclusion and future work are given in the end.

2. ABSTRACT LOGIC TREE

Abstract logic tree has two kinds of nodes: *control nodes* and *task nodes*. The types of control nodes of ALT are used to represent the traverse order of its sub-nodes. There are eight types of control nodes in ALT, including *Root*, *Sequence*, *Branch*, *ASAJ*, *ASOJ*, *Loop Entry*, *Call*, and *Communication*. A *Type* attribute is used to represent the type of each control node. In an ALT, every control node is identified with a unique identifier. Some control nodes have another attributes except for type and identifier.

The notations of control nodes and its informal semantics are given in table 1.

Except for the control nodes, ALT has another kind of nodes, *Task*. Task nodes appear as the leaves in an ALT. Task nodes are represented with a round corner rectangle. As an example, we give one part of the ALT in figure 3 which is transformed from the business process model in figure 2. For brevity, some node identifiers of the ALT are omitted here, and some sub-trees are not completely given in this figure. While introducing how to transform a business process model to an ALT in section 4, the whole ALT of it will be given.

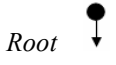
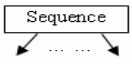
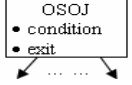

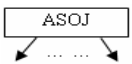
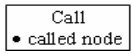
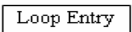
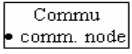
Element	Description
 <p><i>Root</i></p>	<i>Root</i> control node represents the root of an ALT. Root is the entrance of traversing an ALT, and it is also the end point of traversing the ALT.
 <p><i>Sequence</i></p>	<i>Sequence</i> control node refers to traverse its branches from left to right. Once traversing of the rightmost branch is finished, the traverse for this sub-tree finishes.
 <p><i>Or-Split & Or-Join</i></p>	Besides a <i>Type</i> attribute, each <i>OSOJ</i> control node includes a <i>condition</i> attribute that is a conditional expression and an <i>exit</i> attribute which is set of pairs. If the value of conditional expression is represented with <i>val</i> , the form of the pair is (val, id) , where <i>id</i> denotes the identifier of a node in an ALT. According to the computation result of the <i>condition</i> expression and <i>exit</i> attribute, which branches of the control node is traversed can be decided. Once the traversing of a branch is finished, the traverse for the Branch sub-tree is end.
 <p><i>And-Split & And-Join</i></p>	<i>ASAJ</i> control node represents that all the branches of this node must be traversed in parallel. Only when the traversing of all the branches is finished, the traverse for the ASAJ sub-tree finishes.
 <p><i>And-Split & Or-Join</i></p>	<i>ASOJ</i> control node represents all branches of this node must be traversed in parallel. Once the traversing of either branch is finished, the traverse for the sub-tree finishes.
 <p><i>Call</i></p>	<i>Call</i> control node is used to indicate which sub-tree should be traversed next. It is a leaf node of an ALT, and it has a <i>call</i> attribute which indicates the identifier of a node.
 <p><i>Loop Entry</i></p>	<i>Loop Entry</i> indicates which node of the sub-tree should be firstly be traversed. Loop Entry control node represents the entry of a loop, and it often appears together with a Call control node in an ALT.
 <p><i>Communication</i></p>	<i>Commu</i> control node denotes the communication relationships among different control nodes. Generally it is a branch of Sequence control node. It has a <i>commu</i> attribute whose value is an identifier of another <i>commu</i> node, and this identifier indicates which control nodes should communicate with this one. If a <i>Commu</i> control node is met during traverse, the value of its <i>commu</i> attribute is read, and a message is sent to the control nodes indicated by its <i>commu</i> attribute. Once one <i>Commu</i> control node receives the message sent by the other <i>Commu</i> control node, traverse for the current sub-tree stops, and the traverse returns to its parent.

Table 1. Control Nodes of ALT

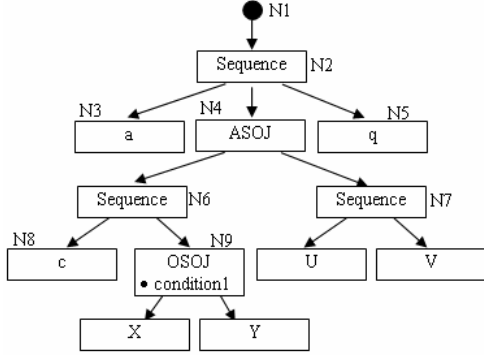


Figure 3. An Example of an ALT

The root of ALT in figure 3 is N1. N1 is the entry of traversing the ALT. It indicates the next traverse object, N2. N2 is a Sequence control node, and it has three branches, N3, N4, and N5. These three branches are traversed one by one from left to right. First, N3 is traversed. N3 is a Task node, and then it is directly traversed. Secondly, N4 is traversed. N4 is an ASOJ control node; thereby its two branches are traversed in parallel. Once traversing for one of these two branches are finished, the traverse for N4 finishes. The traversing for these two branches is not introduced in detail for simplicity. Thirdly, N5 is traversed. N5 is a Task node, and it is directly traversed. Once traversing for N5 is finished, the traverse for N2 finishes. The traverse returns to the parent of N2, the Root node, the traverse finishes.

3. CAPTURE THE ESSENTIAL STRUCTURE OF BPM

ALT provides some control nodes that can be used to represent the execution order of activities of BPMs. For example, sequence execution order can be represented with Sequence control node. Therefore, we need to detect the execution order of activities. According to the definitions of control nodes in ALT, we have five types of the execution order of activities: sequence, branch/or-split-or-join, and-split-and-join, and-split-or-join, and loop. Because loop structure is very complex to be detected, it will be the main part of this section. The other structures detecting will be introduced in brief.

3.1 Loop structure

A BPM may include complex loop structures. Some researchers have studied how to identify the loops for several years [8]. Although the notations of BPM are different from those of flowchart, both of them represent execution order of activities. We simplify the original notations of BPM as the notations of flowgraph [9]. A flowgraph is a connected, directed graph $G=(N, E, START, END)$, where N is the set of nodes, E is the set of edges, $START \in N$ is a distinguished start node with no incoming edges, and $END \in N$ is a distinguished end node with no outgoing edges. For example, the BPM in Figure 2 can be translated into the flowgraph of Figure 4. Every OSOJ element of BPM has an activity to compute the condition expression.

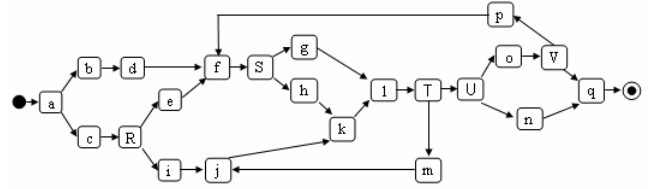


Figure 4. The Simplified Graph of BPM in Figure 2

Since the notations of flowgraph of Figure 4 are different from those of BPM in Figure 2, they represent the same execution order of activities. Therefore, some analysis techniques for flowgraph can be applied to BPM as well. One classical technique for detecting loops is using Tarjan's interval algorithm [11]. The Tarjan intervals are single entry, strongly connected subgraphs [12]. However, Tarjan's interval finding algorithm does not directly handle flowgraphs containing loops with more than one entry, i.e. loops with multiple entries. Based on the number of entries of a loop, loops are divided into two kinds: *general loop* and *irreducible loop*. If a loop only has an entry, it is called a general loop; otherwise it is called an irreducible loop. If a flowgraph includes irreducible loops [13], it is called *irreducible flowgraph*. If a flowgraph doesn't include irreducible loops, it is called *reducible flowgraph*.

The graph in figure 5 includes irreducible loops, for example, $loop0=(j, k, m, f, S, g, l, T, U, o, V, p)$. According to the definition of loop entry, $loop0$ has two entries: f and j . In [9], **DJ graph** was put forward to identify irreducible loops. Before introducing what is DJ graph, a few concepts are introduced firstly. In a flowgraph, node x **dominates** y if and only if all paths from START to y pass through x , denoted by $x \text{ dom } y$. Node x strictly dominates y if and only if $x \text{ dom } y$ and $x \neq y$, denoted by $x \text{ stdom } y$. A node y is said to immediately dominate node x , denoted as $y=\text{idom}(x)$, if $y \text{ stdom } x$ and there is no other node z that $y \text{ stdom } z \text{ stdom } x$. The dominance relation is reflexive and transitive, and can be represented by a tree, called the **dominator tree**.

The *DJ graph* is a flowgraph consists of the same set of nodes as in the flowgraph, and two types of edges called D edges and J edges. D edges are dominator tree edges. The J edges are defined as follows:

Definition (J edges): An edge $x \rightarrow y$ in a flowgraph is named a join edge (or J edge) if $x \neq \text{idom}(y)$. Furthermore, y is named a join node.

Given a DJ graph we distinguish between two types of J edges: Back J(BJ) edges and Cross J(CJ) edges. A J edge $x \rightarrow y$ is a BJ edge if $y \text{ dom } x$; otherwise it is a CJ edge.

Lemma [7]: A flowgraph is irreducible if and only if there exists a simple cycle in its DJ graph that does not contain a BJ edge (that is, the cycle is made of only D edges and CJ edges).

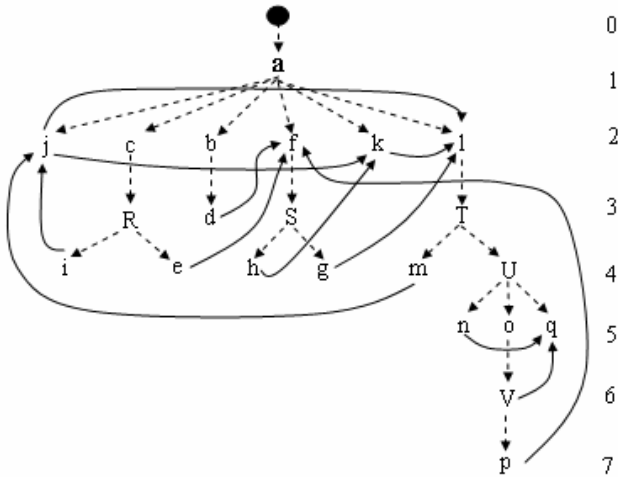


Figure 5. DJ Graph of in Figure 4

In this paper, we will identify general loops and irreducible loops using DJ graphs. The DJ graph of the flowgraph in Figure 4 is given in Figure 5.

According to the above lemmas, loop0 can be easily detected because it doesn't include BJ edges. Using the algorithm for identifying loops given by the authors of [9], both reducible and irreducible loops in a flowgraph can be correctly identified. Furthermore, we can decide which vertexes are the entry nodes of irreducible loops through the following lemma [9]:

Lemma: All the entry nodes of an irreducible loop have the same immediate dominator.

However, the algorithm in [9] doesn't support identification of nested irreducible loops. We extend the original algorithm to identify nested irreducible loops with the following steps:

- 1) For each irreducible loop L do
 - /* If a loop is an irreducible, and detect all of its entry nodes X .
- 2) For each $x \in X$ do
- 3) Delete all the income edges of x
- 4) /*The remain edges and vertexes compose a sub graph G of the original graph.
- 5) To set up the DJ graph of G .
- 6) Using loop identification algorithm [9] to detect both reducible and irreducible loops.
- 7) end
- 8) end

For the flowgraph in figure 4, all the loops of this graph can be detected with the above algorithm. We use bold lines to highlight loops, and use solid rectangle to point out all of detected loop entries in Figure 6.

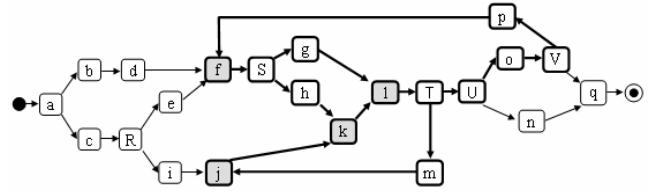


Figure 6. Detected loop entry of BPM in Figure 5

3.2 ASAJ and ASOJ structure

In BPM, ASAJ and ASOJ separately correspond to the following two diagram structures:

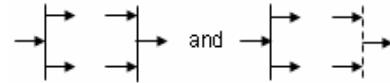


Figure 7. ASAJ and ASOJ structure

Each AND-Split should match an OR-Join or AND-Join. If an AND-Split matches an AND-Join, it is an ASAJ structure. If an AND-Split matches an OR-Join, it is an ASOJ structure. If the join activity is END, the AND-Join or OR-Join usually are omitted in the BPM. Therefore, we need to add the join notation through normalizing the BPM. For example, starting from $x1$ in Figure 2, we detect the OR-Joins, $x2$, $x3$, $x4$, $x5$, and $x6$, and to decide which one is the matched OR-Join to $x1$. Considering $x2$, $x3$, $x4$, and $x5$ are in a loop, it is impossible to be the match of $x1$ because the exits of the loop are not unique. While detecting $x6$, it is easy to find that all the paths starting from $x1$ are merged into a path through $x6$. Therefore, $x1$ and $x6$ compose an ASOJ structure.

The AND-Split is allowed to be nested, which makes the analysis more complex. The following is an example of nested AND-Split:

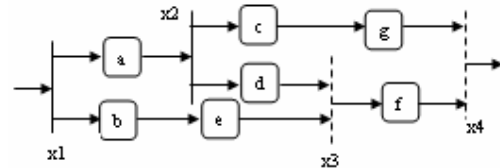


Figure 8. Nested AND-Split

In this example, $x2$ is nested into $x1$, therefore, the detection algorithm will support detecting nested structure. For simplicity, detecting algorithm is not introduced in this paper.

3.3 OSOJ and Sequence structure

In BPM, OSOJ structure is like the following diagram:

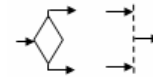


Figure 9. OSOJ structure

The OR-Join is allowed to be omitted if the joint activities of different paths of branch are END, which is a special situation. This special situation can be normalized. The Branch structures are allowed nested. The detecting algorithm is the same as that of detecting ASAJ and ASOJ structure. Sequence structure is easy to be analyzed, and then it is not introduced in detail here.

4. TRANSFORMING BUSINESS PROCESS TO ALT

In order to capture the structure of a BPM, we give the following graph transformation [14] rules.

Using these rules, a BPM can be transformed into an ALT. Furthermore, a BPM may be transformed into different ALTs while using Rule7. To verify the correctness of the transformation, we need to prove that the BPM is structure equivalence with the ALT.

Rule 1:	
Rule 2:	
Rule 3:	
Rule 4:	
Rule 5:	
Rule 6:	
Rule 7:	

Table 2. Transformation Rules

Theorem: Using the transformation rules from BPM to ALT, a Business Process Model can be transformed into a structure equivalent Abstract logic Tree.

Proof: The simplest situation is that a BPM only has a Start node. Obviously, by using Rule1, it can be transformed into a structure equivalent ALT. Both the BPM and ALT have zero paths.

Suppose that a BPM D1 with n elements can be transformed into a structure equivalent ALT T1. And suppose the BPM has x paths, then the corresponding ALT has the same x paths because of structure equivalent. D1 and T1 can be represented as following.

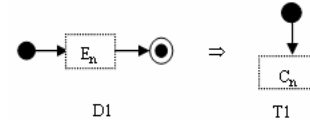


Figure 10. Proof (1)

Once an element is added to the BPM, the new BPM will contain more than n+1 elements. If we can prove that the new BPM can be transformed into a structure equivalent ALT using the transformation rules, the lemma can be proved through induce approach because the graph transformation rules are bidirectional. The added element can be considered according to the following situations:

1) *To add an atomic Task.* If a Task A is added to D1, it is necessary to add a sequence flow. The BPM can be represented by D1' in Figure 11. Using Rule2, the D1' can be transformed into T1'. D1' and T1' still have x paths except that every path is added to a Task A. D1' and T1' are structure equivalent.

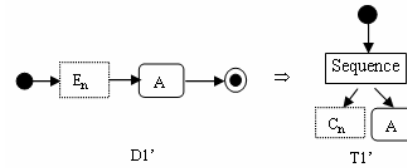


Figure 11. Proof (2)

2) *To add an AND-Split element.* For simplicity, we suppose any BPM has a unique END element. Therefore, an AND-Split element must be matched with an AND-Join or OR-Join element. The result of added element is D1' and D1'' as following through using Rule3 and Rule4, they can be transformed into structure equivalent ALT T1' and T1''.

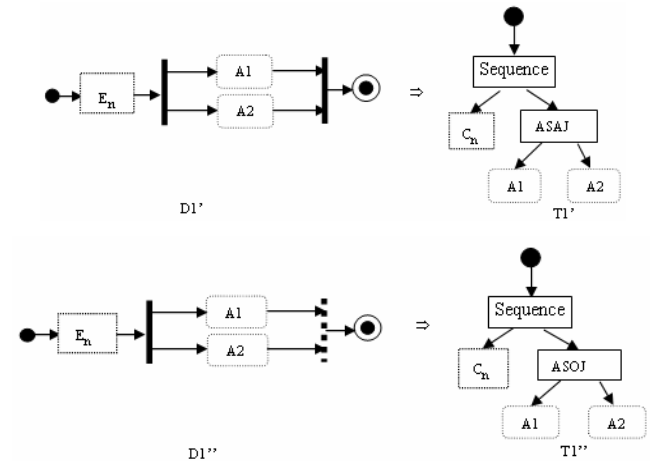


Figure 12 Proof (3)

3) *To add a Branch element.* It will lead to two situations: one situation is a Branch element is match with an OR-Join element, the result is supposition D1'. This situation is simple because

Rule5 can be directly applied, and the structure equivalent CT T1' can be derived. D1' and T1' are shown as following.

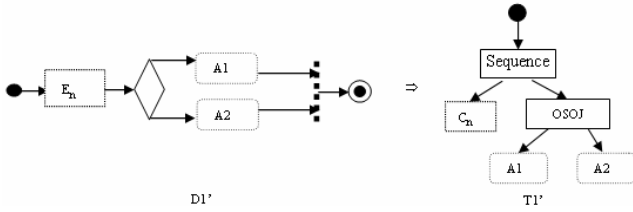


Figure 13. Proof (4)

The other situation is a Branch element adds loops to D1, the result is supposition D2'' (we only consider the situation that a loop is added, the more complex situations can be solved similarly).

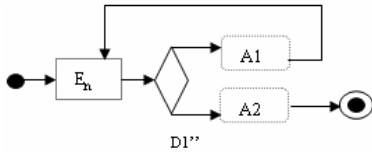


Figure 14. Proof (5)

The entry from A1 to En is denoted by X. According to the structure of En, we can discuss the different situation according to the following classification:

X clearly separate En into two sub-processes, denoted by En1 and En2. When a loop is generated by introducing an element, Rule6 can be applied. Then the ALTs of T1 and T1' can be separately represented as the following.

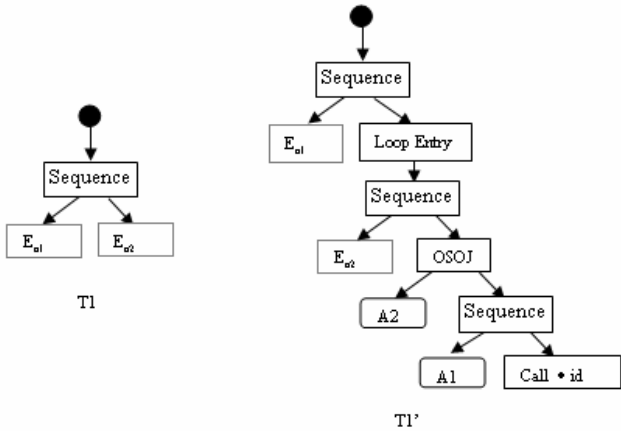


Figure 15. Proof (6)

b.) X may become the other entry of an existing loop, which leads to an irreducible loop. In section 4.1, we have introduced how to identify irreducible loops. Once irreducible loops are identified, Rule6 can be used to transform the BPD to a structure equivalent CT.

Through the above proof, we can induce that any BPM can be transformed into a structure equivalent ALT using the transformation rules.

Based on the above introduction, the structure of a BPM can be represented by an ALT. The corresponding ALT for the BPM in Figure 2 is shown in Figure 16.

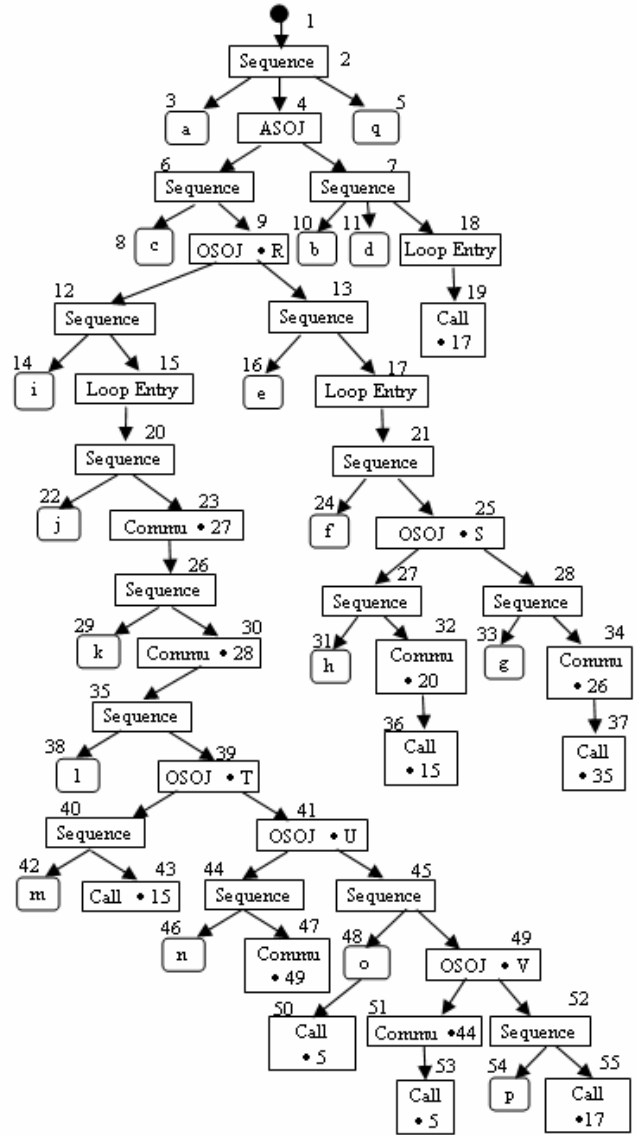


Figure 16. Complete ALT of BPM in Figure 2

5. CONCLUSION

In this paper, we introduced how to capture the essential structures of business process models with abstract logic tree. We defined some graph transformation rules to help transform a BPM into an ALT. We proved that the structure of the BPM and ALT are equivalent. How to identify the loop structures of a BPM is one of the key points for analyzing the structure of a BPM. We extended the algorithm of identifying loops using DJ graphs, which can not only identify the loop structures of a business process model, but also optimize the business process.

Various approaches to business process structure analysis and verification can be found in literatures [15]. However, all these researches employ ad-hoc approaches to analyze the structure of

business process model. In this paper, we proposed to capture the essential structures of business process models with abstract logic trees.

Representing the structure of BPM with ALT has three major advantages: firstly, the structure of BPM can be represented with an isomorphic structure. This isomorphic structure can be implemented with a simple environment to manipulate process models based on tree structure. Secondly, the transformation approach provides a foundation and reference for transforming an unstructured business process to a structured process. Thirdly, a human-readable form of ALT is useful for debugging or analyzing process models. Fourthly, this idea of ALT initially comes from the concept of AST in programming domain. Some analysis technologies built on AST can be leveraged by ALT to analyze business process models.

6. REFERENCES

- [1] BPMI.org and Assaf Arkin. Business Process Modeling Language. 30 Jan. 2003
<http://www.bpmi.org/bpml_prop.esp>.
- [2] BPEL4WS, <http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/>
- [3] OMG Architecture Board MDA Drafting Team, "Model-Driven Architecture: A Technical Perspective", <ftp://ftp.omg.org/pub/docs/ab/01-02-01.pdf>
- [4] Alfred Aho, Ravi sethi and Jeffrey Ullman. Compiler, Principle, Techniques and Tools. Addison-Wesley 1986.
- [5] Joel Jones. Abstract Syntax Tree Implementation Idioms. The 10th Conference on Pattern Languages of Programs, Sep. 8th-12th, 2003.
- [6] ANSI x 3.5 and ISO 5807: 1985
- [7] Rumbaugh, J., Jacobson, I., Booch, G., The Unified Modeling Language Reference Manual, Addison-Wesley, 1998.
- [8] W. Sadiq and M.E. Orłowska. Applying graph reduction techniques for identifying structural conflicts in process model. In Proceedings of the 11th International Conference on Advanced Information Systems Engineering. Heidelberg, Germany, Lecture Notes in Computer Science 1626. PP. 152-209. Springer-Verlag(1999).
- [9] Vugranam C. Sreedhar, Guang R. Gao, and Yong-fong Lee. Identifying loops using DJ graphs. ACM Transactions on Programming Languages and Systems, 18(6):649-658, November, 1996.
- [10] Jay Cousins and Tony Stewart. What is business process and why should I care? White paper. Aug. 2002.
<http://www.rivcom.com/resources/RivCom-WhatIsBPD-WhyShouldICare.pdf>.
- [11] Michael Burke. An Interval-based approach to exhaustive and incremental interprocedural data flow analysis. ACM Transactions on programming Languages and Systems, 12(3):341-395, July 1990.
- [12] R.Bloem, H.N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In W. A. Hunt, Jr. And S. D. Johnson, editors, Formal Methods in Computer Aided Design, PP. 37-54, Springer-Verlag, November 2000. Lecture Notes in Computer Science 1954.
- [13] P. Havlak. Nesting if reducible and irreducible loops. ACM Transactions on Programming Languages and Systems, 19(4):557-567, July 1997.
- [14] G. Rozenberg(ed.), Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. World Scientific, 1997.
- [15] Wasim Sadiq and Maria E. Orłowska, Analyzing process models using graph reduction techniques. Information Systems, Vol. 25, No. 2, PP. 117-134, 2000.