Developing a Workflow Design Framework Based on Dataflow Analysis

Sherry X. Sun
Department of IS
City University of Hong Kong
Kowloon, Hong Kong SAR
sherry.sun@cityu.edu.hk

J. Leon Zhao
Department of MIS
University of Arizona
Tucson, AZ 85721
jlzhao@email.arizona.edu

Abstract

One important step in workflow automation is to design a valid and efficient workflow model that accurately specifies the execution sequence of activities in a business process. The dataflow constraints, which define the data input and output of each activity in a business process, can be used to analyze the dependencies among different activities and determine a proper workflow model for a particular business process. The basic idea of applying dataflow analysis to workflow design has been suggested in the literature. In this paper, we present a framework for applying dataflow analysis to workflow design. Our contribution is threefold. First, we introduce several new instruments, such as activity dependency tree and condition-action table, for specifying the necessary workflow requirements. Furthermore, we develop various key algorithms for requirements analysis and workflow model generation. Third, a system architecture is provided as a roadmap for developing a dataflow-based workflow design tool.

1. Introduction

The technology of workflow automation has been developed to facilitate organizational coordination and collaboration through automating entire work processes and controlling the flow of information among participants [6]. A workflow system can be used to define the work process, control the invocation of activities, route the relevant documents to the appropriate agents, enforce deadlines, and monitor the progress of work. In spite of the promise of workflow automation, many challenges exist. One of them is to design a detailed workflow model that accurately specifies the details of control flow, i.e., the execution sequence of activities and their coordination, in a process.

The literature has suggested that workflow design can start with dataflow analysis [7, 8]. Dataflow specifies what data are needed as input by activities and what data are produced as output by activities. Given that the invocation sequence of activities in a workflow is constrained by the dataflow among activities [3], the dataflow constraints can be used to analyze the dependencies among different activities and determine a proper workflow model for a particular business process.

The dataflow-analysis-based workflow design starts with identifying the set of activities needed to be included in a workflow and their input and output data through analyzing existing business processes and paperwork, e.g., forms, documents, and product specifications [4, 7, 8]. Then the control flow can be determined from the dataflow using the following principle. The activity producing a data item as output has to be sequenced before the activities using that data item as output [7, 8]. For example, in an order processing workflow, the activity *update inventory* uses the product quantity ordered by a customer as input to decide the new inventory level. Therefore, the activity *process order*, which produces the output "product quantities ordered", must be executed before the activity *update inventory*.

Even though the formal approach of dataflow-analysis-based workflow design reported in the literature provides a theoretical foundation, the challenge still remains in transforming the formal approach into design tools that can be used in practice. In this paper, we present a framework that serves as a roadmap for applying dataflow analysis to workflow design in practice. Our contribution is threefold. First, we introduce several new instruments, such as activity dependency tree and condition-action table, for specifying the necessary workflow design requirements. Furthermore, we develop various key algorithms for requirements analysis and workflow model generation. Third, a system architecture is provided as a roadmap for developing a dataflow-based workflow design tool.

The rest of this paper is structured as follows. Section 2 briefly reviews the relevant literature. Section 3 presents a workflow design framework. Section 4 describes the tools and the steps for requirements collection in workflow design. Section 5 presents the algorithms for requirement analysis. Section 6 discusses a procedure for generating workflow models. A system architecture is shown in Section 7. Finally, Section 8 concludes the paper by highlighting our contributions and outlining future research directions.

1530-1605/08 $25.00 © 2008 IEEE

2. Literature Review

The traditional workflow design approach, referred to as participative approach [2], adopts the principle of joint application design (JAD), which describes a variety of methods for conducting workshops where users and technical developers work together to define requirements of information systems [1]. The participative approach is useful for collecting the needed information for workflow design. However, the participative approach does not provide any formalism for generating a workflow model with the information that has been collected.

The workflow development methodologies found in the literature rather focus on representing workflow models appropriately and customizing the traditional software development approach to meet the specific needs in workflow development, than emphasizing on the principles of creating workflow models [3, 10]. Various criteria have been proposed from different perspectives to help determine whether a workflow model is correct [5, 11]. However, applying those criteria still requires design methods to create the initial workflow models.

The product-based workflow design [4] and the policy-driven workflow mapping method [9] have been developed in order to address the challenge of generating workflow models systematically based on the information collected from exiting documents. In the product-based workflow design, breadth first search and depth first search are used to identify the workflow models that follow the relationship among data elements derived from product specifications [4]. In the policy-driven workflow mapping method, design rules are developed to help derive workflow model from narrative business policies [9].

The basic idea of designing workflow models based on dataflow analysis was proposed in [7]. Moreover, formal design rules have been developed to help sequence two regular activities and place routing activities, i.e., ANDSplit, ANDJoin, XORSplit, and XORJoin, through dataflow analysis [8]. Those formal design rules provide the foundation to automate the workflow design process. This paper extends the existing theory in a practical direction by providing a detailed framework, tools, and algorithms for applying the dataflow-analysis-based workflow design in practice.

3. A Workflow Design Framework Based on Dataflow Analysis

This section presents a framework for implementing the dependency based workflow design. As shown in Figure 1, this framework consists of three major steps: 1) requirements collection, 2) requirements analysis, and 3) workflow design. The details of each step are described in the following sections. Note that UML activity diagrams are used in all the illustrating examples.

4. Requirements Collection

This step aims to answer two questions: what activities should be included in a workflow model? What are the input and output data for each activity? To answer these two questions, we first determine the overall goal of a workflow process, outline the set of activities that can be organized to achieve the goal, and then identify the input data and output data for each activity. Moreover, we analyze the business rules relevant to process routing and refine the set of activities according to the business rules.

4.1. Analyze Business Goal and Data and Activity Dependencies

From the data processing perspective, the goal of a workflow is to transform the set of input data available at the beginning of a workflow to the set of output data in need. For example, an insurance claim process takes customer's name, insurance policy number, damage cost as input and generates a data output that states the claim is either approved or rejected. As such, the initial input data and the final output data need to be decided at the beginning of the workflow design process. The initial input data and the final output data for a workflow process can be identified through collecting and analyzing the existing paperwork, including forms, documents, and product specifications, and having meetings and interviews with people involved in the existing workflow process. In case that a process starts without any initial input data set, the first activity that produces the initial data set needs to be identified first.

Once the initial input and final output data are known, the set of activities, which transform the initial input data into the final output data, can be traced in three steps: 1) The set of activities V that directly produce the final output data is identified first; 2) If any activity in V needs an input data d neither provided by the initial input data set nor produced by any other activities in V, the activity producing data d as output needs to be identified and included in V; 3) repeat step 2) until no more activities can be added into V.

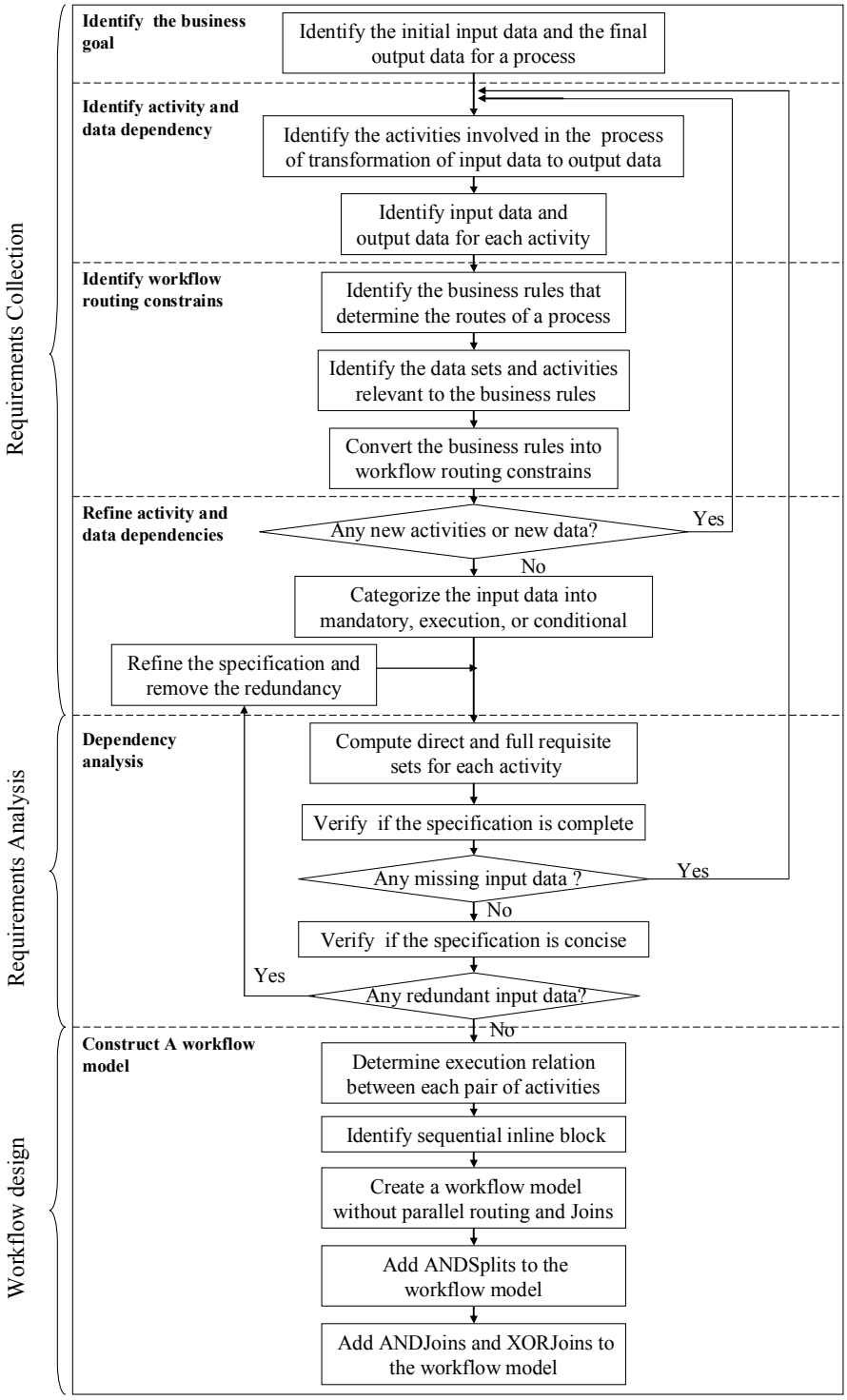


Figure 1. A workflow design framework based on dataflow analysis

An activity dependency tree can be used to facilitate the identification of the set of activities that need to be included in a workflow model.

Definition 1 (Activity dependency tree) An activity dependency tree T is a graphic structure where the root node represents the end activity (i.e., the last activity in a workflow model), all other nodes represent different

activities that produce some output data used as input by their parent nodes.

An activity dependency tree can be created using the following procedure

(1) create the root node *e*;
(2) find all the set of activities *V* that directly produce the final output data;
(3) create a node for each activity in *V* and attach these nodes as the child nodes of *e* to the tree;
(4) for each activity *v* in *V*, if the input data required by *v* is produced by another activity *u* that does not have a corresponding node in the tree, create a node for *u* and attach the node as a child node of *v* to the tree;
(5) then add *u* to *V*
(6) repeat (4) and (5) until no more nodes can be added into the tree.

It is worth noting that the above procedure to create an activity dependency tree is very similar to the steps we need to identify the set activities to be included in a workflow model. An activity dependency tree essentially provides a visual tool. Furthermore, activity dependency tree can assure that all the identified activities can be linked together through data and activity dependencies. This is important because it guarantees that all the identified activities can be added to one workflow model and they will not form unrelated groups when the design procedure presented later in this paper are applied. Note that when activities *u* and *v* both use some data produced by activity *x*, there can be different activity dependency trees since activity *x* can be the child node of either activity *u* or activity *v*. however, both trees identify the same group of activities.

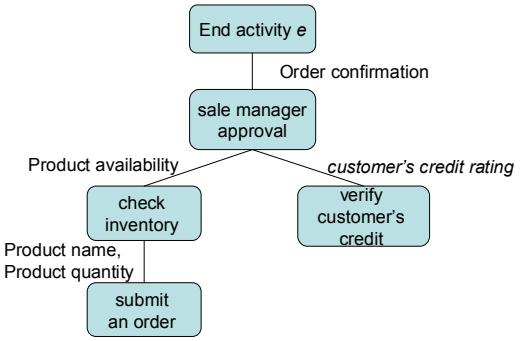


Figure 2. An example of activity dependency tree

We illustrate how to create an activity dependency tree through an order processing example, in which after a retailer receives an order from customer, the retailer processes the order and send an confirmation to the customer. The initial input data for the process include *customer's name*, *product name*, *product quantity*, and *payment method*, which are produced by the activity *submit an order*. The final output is either an approval or a rejection, which is referred to as *order confirmation*. The order confirmation is actually the output data produced by the activity *sale manager approval*, which takes *product availability* and *customer's credit rating* as input. Furthermore, we identify the two activities, *check inventory* and *verify customer's credit*, produce *product availability* and *customer's credit rating* as output respectively. Given that the activity *check inventory* takes *product name* and *product quantity* as input, the activity *verify customer's credit* takes *customer's name* and *payment method* as input, and all these data items are provided as the initial input by the activity *submit an order*, we can create the activity dependency tree as shown in Figure 2.

4.2. Identify Workflow Routing Conditions and Refine Data and Activity Dependencies

Different workflow instances can take different paths under the guidance of business rules. To design a workflow, it is important to collect those business rules and then format them into workflow routing conditions. Business rules can be identified by interviews or extracted from existing policies [9]. To format a business rules in plain English into workflow routing conditions, we can use a Condition-Action table, where a workflow routing condition is expressed as

Condition: Criteria(*D*) *Action:* Execute(*V*)

and *D* is a set of data, and *Criteria(D)* is a logical expression of *D*. The routing condition above means that when the set of data *D* meets certain criteria, the set of activity *V* is executed. Each data item in the data set *D* can be binary, categorical, or numerical data. That is, the criteria for *D* can be written a Boolean expression such as "$x = a$" or "$a \leq y < b$". For example, the business rules in an order processing example (Table 1) can be formatted into the Condition-Action table show in Table 2.

As defined in [7], if the data item is included in a routing condition that determines when an activity should be executed, then the activity has an execution dependency on that data item. Analysis of routing conditions can help refine input and output data and further identify the set of activities to be included in a workflow model. From Table 2, we can identify two more activities for the order processing example, which are not included in the activity dependency tree in Figure 2. They are *send replenish order* and *send back order notice*. The output from these two activities, e.g., *replenishment quantity* and *back order notice status*, should be added to the final output data set produced by the workflow. Then the activity dependency tree in Figure 2 needs to be modified.

Table 1. Business rules for the order processing example

Rule 1	If a customer orders more than what is available, a replenish order should be sent to the manufacturer and a back order notice should be sent to the customer.
Rule 2	If there are enough products in the inventory, the customer's credit will be checked.
Rule 3	If the credit rating of a customer is good, the order needs to be approved by a sales manager

Table 2. The condition-action table for the order processing example

Routing Constraint	Condition	Action
R1	*Product availability* = "*No*"	*Execute*(*send replenish order*, *send back order notice*)
R2	*Product availability* = "*Yes*"	*Execute*(*verify customer's credit*)
R3	*customer's credit rating= "good"*	*Execute*(*sale's manager's approval*)

When a set of activities is well specified, the input data required by each activity can be grouped into three categories, i.e., mandatory, conditional, and execution input data, depending on the roles the data items play as input [7, 8]. Accurate categorization of input data is fundamental to workflow design. The dependency an activity has on the input data set is referred to as data dependency [8].

Further, from the identified data dependencies, activity dependencies can be derived using the following principle: If activity u uses data item d as input and d is the output from activity v, then u depends on v, which is referred to as activity dependency [8].

5. Requirements Analysis

The purpose of this step is to derive information useful for design from the data and activity dependencies identified at requirements collection.

5. 1. Compute Requisite Sets

At this step, in order to further analyze the activity dependencies, the direct and full requisite sets, which are used to determine whether the data dependencies are both complete and concise (see 5.2.), can be constructed for each activity as defined below.

Definition 2 (Direct Requisite Set Δ_v) A set of activities Δ_v is the direct requisite set for activity v if for any activity $x \in \Delta_v$, there exists a data item d such that $d \in O_x$, $d \in I_v$, where I_v is the input data set of activity v, O_x is the output data set of x, and $x \neq v$.

Definition 3 (Full Requisite Set Γ_v) Given a set of activities V, the full requisite set Γ_v for $v \in V$ is a subset of V such that for every $u \in \Gamma_v$ the invocation of v directly or indirectly requires some output data produced by u.

Essentially, the direct requisite set Δ_v for activity v is the set of activities that produce the data directly used by v as input. The full requisite set Γ_v for v is the set of activities that transform the external input data available at the various activities of a workflow into the data needed by v as input. Some of the activities in Γ_v may not produce data used by v directly as input but provide some intermediate data needed to produce the data items in I_v. Note $\Gamma_v \supseteq \Delta_v$.

The procedure Construct_Direct_Requisite_Set shown in Figure 3 can help to automatically create direct requisite sets for a set of activities. For each data item d needed as an input by an activity, the procedure tries to find the activity that produces data item d as an output and create a dependency link between these two activities.

In order to create the full requisite sets, the procedure Construct_Full_ Requisite _Set shown in Figure 4 starts with a set of activities and their direct requisite sets and applies a depth first search in four steps. For each activity v, 1) the procedure marks the activities in the direct requisite set of v as identified and push them into a stack; 2) An activity u is popped from the stack and the procedure examines the direct requisite set of u if u has a non-empty direct requisite set; 3) any activity z in the direct requisite set of u is marked as identified and pushed into the stack if z has not been marked as identified; 4) repeat the steps 2) and 3) until no more activities can be obtained from the stack.

```
Procedure Construct_Direct_Requisite _Set
 Input:  A set of activities V and the input data set and
  output data set of each v in V
 for each activity v in V
  for each input data d required by v
      activity
   temp=findOutputActivity(d)
 //Find the activity that produces d as output
  if (temp≠null)  then
// the activity producing d as output can be found
  add temp to the direct requisite set of v
```

Figure 3. The algorithm of constructing direct requisite sets for a set of activities

5. 2. Completeness and Conciseness

Given the set of activities and their data dependencies identified at the previous step, we need to make sure that adequate but succinct information has been collected for workflow design. Two measures are used to verify the correctness of the specification for workflow design, i.e., completeness and conciseness.

```
Procedure Construct_Full_Requisite _Set
 Input:  A set of activities V and the direct requisite
  set of each v in V
 for each activity v in V
      create a stack
      for activity v in V
                  set identified(v) = false
      for each activity u in the direct requisite set of v
   //initialize the stack
      set identified(u) = true
        stack.push(u)
          while (stack is not empty) do
           u=stack.pop()
           if(direct requisite set of u contain
              activities) then
            for each activity z in the direct requisite
                set of u
             if(not identified(z)) then
                stack.push(z)
                set identified(z) = true
              else add u to fullRequisiteSet(v)
```

Figure 4. The algorithm of constructing full requisite sets for a set of activities

Completeness computes if the set of activities can produce the input data needed in the workflow. Conciseness determines if the set of activities produce more output data than needed. We define the two measures as follows. Note we use I_v and O_v to denote the input and output data sets of activity v, respectively.

Definition 4 (Completeness of Dataflow) Given a set of activities V, if $\forall v \in V$ and Δ_v, there exists u such that $u \in \Delta_v$ and $d \in O_u$ for $\forall d \in I_v$, then we say the dataflow specification of V is complete.

```
Procedure Verify_Completeness
 Input:  A set of activities V and the direct requisite
  set of each v in V
 for each activity v in V
  for each input data d required by v
   //mark the activity producing d as unknown for
     //each activity v in V
  for each input data d required by v
   if (outputActivity(d) = "unknown") then
       activity temp=findOutputActivity (d,
       directRequisiteSet(v))
       //Find the activity that produces d as output //from
       the direct requisite set of v
     if (temp=null)  then
       set  outputActivity(d) = "none"
       // the specification is not complete
```

Figure 5. The algorithm of verifying completeness

Definition 5 (Conciseness of Dataflow) Given a set of activities V, the dataflow specification of V is concise if the following two conditions hold: 1) for each $d \in O_{vi}$, $v_i \in V$, there exists $v_j \in V$ such that $d \in I_{vj}$; and 2) for each $d \in I_{vi}$, $v_i \in V$, there exists only one $v_j \in V$ such that $d \in O_{vj}$.

We provide two algorithms to automate the step of requirements analysis. The procedure Verify_ _Completeness shown in Figure 5 decides if the set activities can produce all the required input data, i.e., the completeness of the set of activities. At the beginning, the procedure labels the activities that produce the required input data as unknown. Then the procedure examines the direct requisite set of each activity and changes the corresponding label to none if the activity producing a required input data cannot be found.

```
Procedure Verify_Conciseness
 Input:  A set of activities V , the input data set and
  output data set of each v in V,  and the final output O
 for each activity v in V
 for each output data d_v required by v
   activity temp=findInputActivity(d_v)
   //Find the activity that uses d as input
   if (temp=null and d_v is not included in O)
   //the activity producing d as output can be found
   then  print "no activity uses d as input"
   for each activity u in V and u≠v
     for each output data d_u required by u
       if (d_v is the same as d_u)
         //two activities produce the same data
         then print "d_v produced by v is the same as d_u
           produced by u"
```

Figure 6. The algorithm of verifying conciseness

The procedure Verify_Conciseness shown in Figure 6 first examines whether each output data item has been required either as input for an activity or the final output from a workflow. Then the procedure compares all the output data and determines if a data item has been produced multiple times. An error message is printed out if an output data does not contribute to the production of final output data or a data item is produced more than once.

If a specification for workflow design is not complete or concise, the step of requirement collection may be repeated in order to modify the specification, identify the activities that have been missed, and remove the redundant activities that produce useless data.

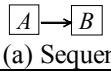
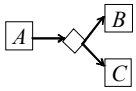
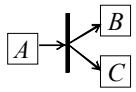
6. Workflow Design

This section discusses how to generate a workflow model from activity and data dependencies resulted from the requirements collection and requirements analysis.

6. 1. Identification of Activity Relations

Five basic workflow constructs, i.e., sequence, AND-Split, AND-Join, XOR-Split, and XOR-Join, have been defined as the essential workflow primitives by Workflow Management Coalition (WfMC). In order to create a workflow model, we need to answer a fundamental question, i.e. when a basic construct can be used. Table 3 shows the four types of activity relations, i.e., immediate precedence, conditional precedence, XOR-parallel, and AND-parallel, which need to be decided based on data and activity dependencies at the design step. For the details of deriving the four types of activity relations from data and activity dependencies, please refer to [8].

Table 3. Workflow constructs and activity relations

Workflow Construct	Activity Relation
 (a) Sequence	Immediate Precedence: *A* immediately precedes *B*
 (b) XORSplit	Conditional Precedence: *A* conditionally precedes *B*
	XOR-parallel: *B* XOR-parallels with *C*
 (d) ANDSplit	AND-parallels *B* AND-parallels with *C*[1]

[1] We can consider the activity relation between *A* and *B* as *A* immediately precedes *B*.

6. 2. Identification of Sequential Inline Blocks

WfMC (1999) defined the concept of *inline block* as a collection of activities that has one entry point and one exit point. An inline block may be condensed into a block activity corresponding to a sub-process containing the inline block. In this paper, we consider only sequential inline blocks, which is sufficient for simplification of workflow design. A sequential inline block is a single chain of activities starting with *u* and ending with *v*. In the block, there is no XORSplit, XORJoin, ANDSplit, or ANDJoins. Activity *u* is called the *source* of the block and activity *v* is called the *sink* of the block.

Sequential inline blocks can be created as follows. If we know activity *u* immediately precedes activity *v* and only activity *v* and no other activities immediately precedes *v*, then activities *u* and *v* forms a smallest sequential inline block. Further, if we know *v* immediately precedes activity *z* and only activity *z* and no other activities immediately precedes *z*, then activities *u*, *v*, and *z* form a sequential inline block. Identification of sequential inline blocks can help simplify workflow design given that the set of activity included in the block can be replaced by a block activity.

6. 3. Create a Workflow Model without Parallelism and Joins

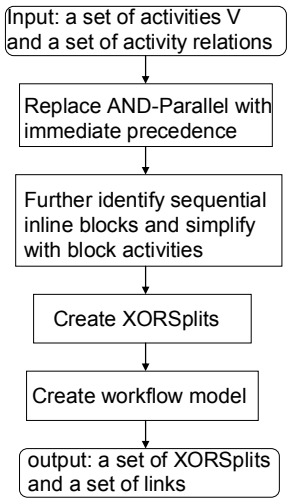
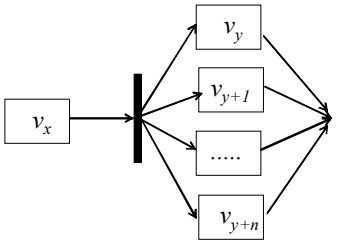


Figure 7. The procedure of creating a workflow model without parallelism and Joins

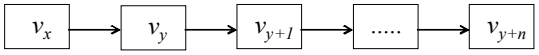
Once the activity relation between each pair of activities has been derived data and activity dependency analysis, it is quite complicated to generate a model from activity relations. To simplify the design procedure, we derive a workflow model in three stages.

In Stage 1, we deal with the correctness of the workflow model by ignoring parallelism. That is, we will randomly sequence activities that could potentially be put in parallel. Since parallelism is used only to improve efficiency, the resulting workflow model without parallelism is correct though not efficient. At this stage, we will treat AND-parallel relations as if they are immediate precedent relations. In Stage 2, we will take into account of the AND-parallel relations that were ignored in Stage 1. In Stage 3, we standardize the model by adding joins such that each business activity can have only one incoming link and one outgoing link.

Figure 7 shows the procedure for creating a workflow model without parallelism and Joins. The first step is to replace the AND-parallel with immediate precedence. If we know v_x AND-parallels with v_y, v_{y+1}, v_{y+2}, …, v_{y+n} as shown in Figure 8A, we can replace the parallel execution with sequential execution. As shown in Figure 8B, the AND-parallel relations between v_{y+i} and v_{y+i+1} is replaced by an immediate precedence relation, i.e., v_{y+i} immediately precedes v_{y+i+1}.



A. Activities in parallel execution



B. Activities in sequential execution

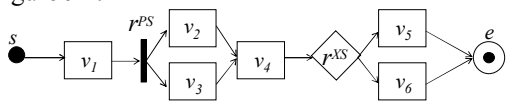
Figure 8. An example for replacing parallelism with sequential execution

Since we ignore the parallelism temporarily, we only need to consider where to use the two basic constructs shown in Table 3, i.e., sequence and XORSplit. Therefore, the resulting workflow model does not have any ANDSplit. Moreover, we do not consider where to add Joins at this step. As such, in the resulting model, a regular activity may have more than one incoming link.

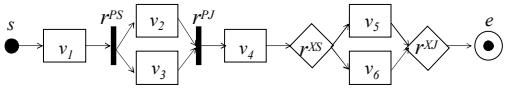
6. 4. Add AND-Splits, AND-Joins, and XOR-Joins

At this step, we first rectify the model by examining the sequential inline blocks resulting from the replacement of AND-Parallel relations and adding AND-Splits back to the model. Essentially, we replace the sequential block shown in Figure 8B back with the parallel structure shown in Figure 8A.

Then we add the XORJoins and ANDJoins to the model. The principles used to add Joins can be explained with the example in Figure 9A. If an activity is preceded immediately by more than one activity, a Join should be added. In Figure 9A, a Join should be added before activity v_4 and the end activity e. To determine whether an ANDJoin or XORJoin is needed, the two different paths from the start activity s to v_4, denoted as $path_1$ and $path_2$, need to be compared. Note $path_1=\{s, v_1, r^{ps}, v_2\}$ and $path_2=\{s, v_1, r^{ps}, v_3\}$. The last activity shared by $path_1$ and $path_2$ is the ANDSplit r^{ps}, indicating that $path_1$ and $path_2$ are two parallel branches. Therefore, an ANDJoin needs to be added before v_4. By the same token, an XORJoin should be added before e, leading to the workflow model shown in Figure 9B.



A. A workflow model without Joins



B. A workflow model with Joins

Figure 9. An simple workflow design example

7. A Component Based System Architecture

7. 1. System Design

Figure 10 gives an outline of the component based architecture for implementing the proposed approach, which consists of five main components: *Interface*, *Dataflow Verifier*, *Dependency Analyzer*, *Workflow Modeler*, and *Design Coordinator*. The functions of these components are discussed below. Note that the arrows indicate the interaction among different components.

- *Web Based User Interface*: an interface is provided for workflow developers to specify the set activities and their input and output data to the workflow designer. The interface sends the error message to workflow developers when the specification is incorrect. A workflow developer can access the interface through a web browser where the final model is displayed as a UML graph. The interface can also be invoked by a SOAP request and then the final model is sent back in a XML encoded file.
- *Dataflow Verifier*: the *Dataflow Verifier* examines

whether the specification sent by a workflow developer, i.e., the set of activities and their input and output data, contains enough information for workflow design. It consists of three modules. The *requisite sets generator module* calculates the direct and full requisite sets for each activity. The *completeness verification module* analyzes the specification to assure all the input data are provided either as the output from some activities or by external resources and there is no missing data. It implements the procedure *Verify_Completeness* (Figure 5). The *conciseness verification module* evaluates the specification for data redundancies, i.e., output data not required and repetitious production of the same data item. It implements the procedure *Verify_Conciseness* (Figure 6).

- *Dependency Analyzer:* The *Dependency Analyzer* analyzes the specification sent by a workflow developer and generates a set of activity relations that represent the possible execution steps in the workflow model. It has three modules: *sequences identifier module* helps determine the activities that can be executed sequentially, including sequential inline blocks; *XORSplit identifier module* and *ANDSplit identifier module* helps decide where to place XORSplit and ANDSplit.

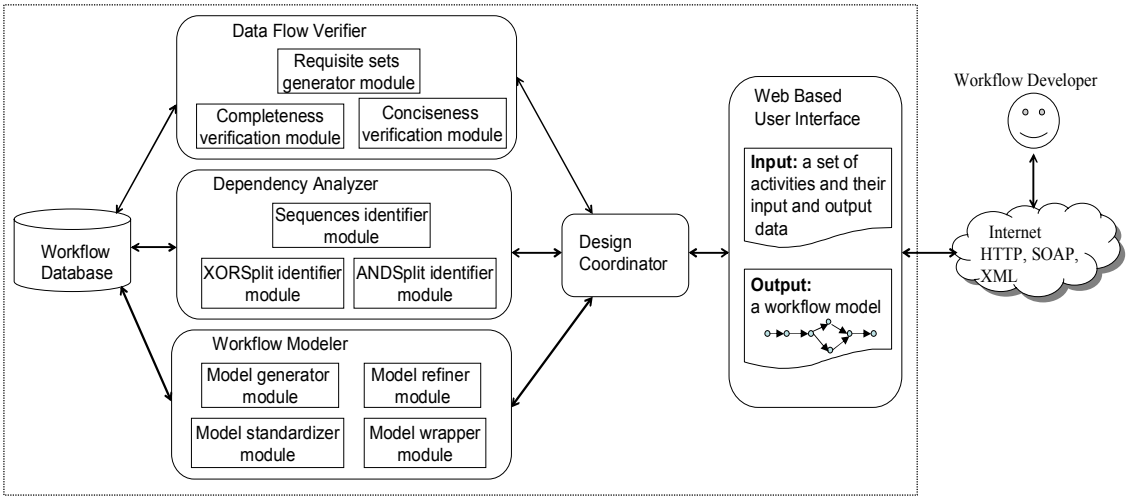


Figure 10. A high level system design of the dependency analysis based workflow designer

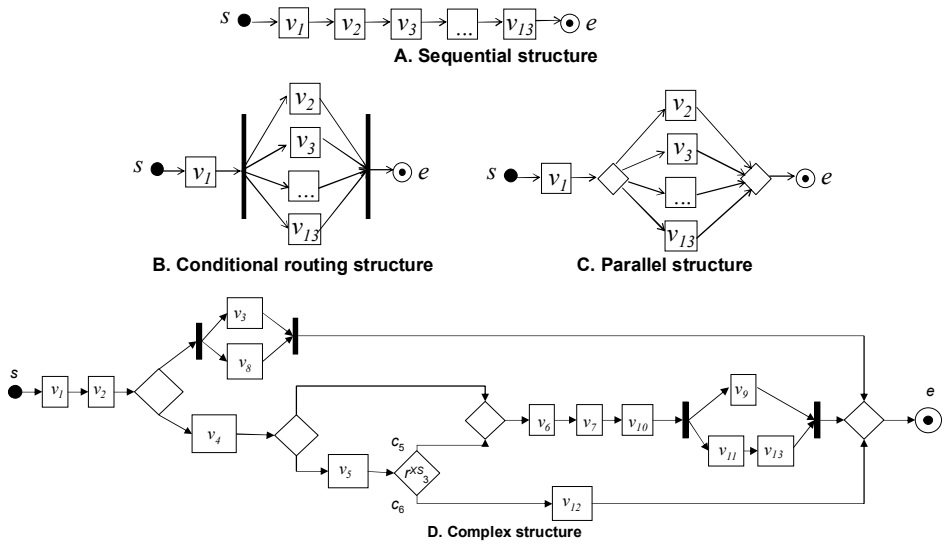


Figure 11. Workflow models tested by the proof-of-concept implementation

- *Workflow Modeler*: The *Workflow Modeler* takes the set of activity relations as input and generates a standard workflow model. The *model generator module* is responsible for creating a basic workflow model without considering parallelism and Joins. The *model refiner module* incorporates parallelism into the workflow model. The *model standardizer module* identifies where ANDJoins and XORJoins are needed. The *model wrapper module* generates a XML document, which describes the structure of a final workflow model.
- *Design Coordinator:* directs the interactions among different components.

7. 2. A Proof-of-Concept Implementation

In order to test the core algorithms for constructing a workflow model from activity relations, a proof-of-concept implementation has been done in Microsoft Visual Basic, an object oriented programming language designed to be easy to use with the flexibility to to develop fairly complex applications. Microsoft Excel spreadsheets are used to store the activty relations. The preliminary implementation shows that the proposed procedure is capable of generating not only the simple workflow models as shown in Figure 11A, 11B, and 11C, but also complex workflow models such as the example shown in Figure 11D.

8. Conclusions

In this paper, we developed a detailed framework for applying dataflow analysis to workflow design including various tools, detailed algorithms, and system architecture. Our research aims at minimizing the difficulties of applying the proposed approach in workflow analysis and design. In order to test the core procedure for constructing workflow models based on dataflow analysis, a proof-of-concept implementation was developed. We are currently developing a prototype system with full functionalities. User studies will be conducted using the prototype system for further validation of the approach of dataflow-based workflow design.

9. References

[1] Davidson, E. J. "Joint Application Design (JAD) in Practice," Journal of Systems and Software, 1999, 45, 3, pp. 215-223.

[2] Herrmann, T., and Walter, T. , "The Relevance of Showcases for the Participative Improvement of Business Processes and Workflow Management," Proc. of Participatory Design Conf., 1998, pp.117–127.

[3] Kwan, M.M., and Balasubramanian, P.R., "Dynamic Workflow Management: a Framework for Modeling Workflows," Proceedings of HICSS 1997, Maui, HI, USA, 7-10 January 1997, vol. 4, IEEE Computer Society Press, pp. 367 – 376.

[4] Reijers, H.A., Limam, S., Aalst, van der W.M.P. "Product-based workflow design," Journal of Management Information Systems 2003, 20, 1, pp. 229-262.

[5] Sadiq, W. and Orlowska, M.E., "Analyzing Process Models Using Graph Reduction Techniques," Information Systems 2000, 25, 2, pp. 117–134.

[6] Stohr, E. A. and Zhao, J. L. , "Workflow Automation: Overview and Research Issues", Information Systems Frontiers, 2001, 33, 281-296.

[7] Sun, S. X. and Zhao, J.L. "A Data Flow Approach to Workflow Design," Proceedings of WITS 2004, Dec 11-12, Washington D.C., 80-85.

[8] Sun, S. X. and Zhao, J.L. "Activity Relations: A Dataflow Approach to Workflow Design," Proc. of the 2006 Intl. Conf. on Information Systems, Dec. 10-13, 2006, Milwaukee, USA.

[9] Wang, H. J., Zhao J. L., Zhang L. J. "Policy-Driven Process Mapping (PDPM): Towards Process Design Automation", Proc. of the Intl. Conf. on Information Systems, Dec. 10-13, Milwaukee, USA.

[10]Weske, M., Goesmann, T., Holten, R., and Striemer, R. "A Reference Model for Workflow Application Development Processes," Proceedings of the International Conference on Work activities Coordination and Collaboration, 1999, pp. 1-10.

[11]Wieringa, R. J., Gordijn, J., "Value-Oriented Design of Service Coordination Processes: Correctness and Trust", 2005 ACM Symposium on Applied Computing, pp. 1320-1327.