CORE Metadata, citation and similar papers at core.ac.uk

Provided by AIS Electronic Library (AISeL)

Association for Information Systems

AIS Electronic Library (AISeL)

ICIS 2004 Proceedings

International Conference on Information Systems (ICIS)

December 2004

Process Logic for Verifying the Correctness of Business Process Models

Henry Bi
Pennsylvania State University

Leon Zhao
University of Arizona

Follow this and additional works at: http://aisel.aisnet.org/icis2004

Recommended Citation

Bi, Henry and Zhao, Leon, "Process Logic for Verifying the Correctness of Business Process Models" (2004). *ICIS 2004 Proceedings*. 8.
http://aisel.aisnet.org/icis2004/8

This material is brought to you by the International Conference on Information Systems (ICIS) at AIS Electronic Library (AISeL). It has been accepted for inclusion in ICIS 2004 Proceedings by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.

PROCESS LOGIC FOR VERIFYING THE CORRECTNESS OF BUSINESS PROCESS MODELS

Henry H. Bi
Smeal College of Business
Pennsylvania State University
University Park, PA U.S.A.
henrybi@psu.edu

J. Leon Zhao
Eller College of Management
University of Arizona
Tucson, AZ U.S.A.
lzhao@eller.arizona.edu

Extended Abstract

Process verification is a key step in business process management. In this paper, we propose process logic as a new logical formalism and mathematical method to enable advanced process verification. We formally define the syntax and semantics of process logic, establish a formal relationship between process logic and graphical representation of process models, and transform the problem of verifying the correctness of process models into a problem of determining the validity of logic argument forms.

Keywords: Process logic, process verification, process graphs

Introduction

Process verification is a key step in business process management. The purpose of process verification is to verify the correctness of process models during design time. Process verification is important since detecting process anomalies before process models are put into operation can help reduce high costs of breakdown, debugging, and fixing during runtime. A process anomaly is simply an improper design that causes execution errors.

In this paper, we propose process logic as a logical formalism and mathematical method to enable advanced process verification. We first define formally the syntax and semantics of process logic. We then establish a formal relationship between process logic and graphical representation of process models. The process logic we propose allows us to transform the problem of verifying the correctness of process models into a problem of determining the validity of logic argument forms.

Literature Review

Several process verification methods have been proposed. Verification based on Petri Nets requires translating process models into Petri nets for indirect verification (van der Aalst et al. 2002). Graph reduction techniques (Sadiq and Orlowska 2000) can detect a limited set of process anomalies because the set of the developed reduction rules is not complete (van der Aalst et al. 2002). The matrix-based workflow verification approach (Choi and Zhao 2002, 2003) has not addressed how to handle process models that contain *OR* relationships among activities that are included in process modeling formalisms (Bi and Zhao 2004b; Mayer et al. 1995; van der Aalst and Kumar 2003).

Transaction logic (Bonner and Kifer 1994; Kifer 1996) is used to analyze state-based process models, in which the notion of states corresponds to the notion of database states. Nevertheless, most process modeling paradigms in existing information systems apply activity-based modeling (Davenport 1993; Lin et al. 2002).

Our previous research has demonstrated the power of propositional logic to verify activity-based process models (Bi and Zhao 2003, 2004a). Furthermore, we have also demonstrated that propositional logic has some limitations in describing process phenomena. This paper extends our previous research results by proposing formally the notion of process logic, which defines the syntax and semantics of process logic in a more precise way to reflect the specific characteristics of process structures. In process logic, we differentiate the definitions of join structures from those of split structures, which cannot be done in propositional logic. We also refine the notations of logical operators *and*, *or*, and *xor* to reflect that they are n-ary ($n \geq 2$) operators in process logic while they are binary operators in propositional logic.

Process Logic

Syntax of Process Logic

Process logic is concerned with the analysis of process arguments regarding process models.

Definition 1 (process argument). A *process argument* is a sequence of process propositions, in which one is intended as a *conclusion* and the others, the *premises*, are intended to prove the conclusion. ■

For instance, the following argument is regarding a simple process model represented with a standard process graph (see Appendix A) in Figure 1:

(1) After the start vertex s is executed, a control vertex c_1 is activated.
(2) The start vertex s is executed.
∴ (3) The control vertex c_1 is activated.

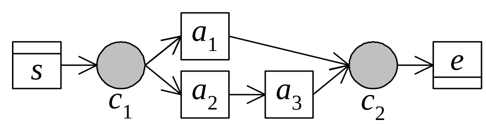

Figure 1. A Standard Process Graph P_S

In this process argument, (1) and (2) are premises, and (3) is the conclusion. (2) and (3) are simple process propositions, and (1) is a process proposition. The symbol ∴ means therefore.

Definition 2 (language of process logic). The language that consists of symbolic notation to represent process models is called the *language of process logic*. ■

The language of process logic is described in two steps: symbols of the language and formulas of the language.

Definition 3 (symbols). The *symbols* of the language of process logic are

(1) Process variables: *Process variables* are interpreted as simple process propositions and are denoted by lowercase letters with or without numerical subscripts: a, b, c, …, a_1, a_2, a_3, …, b_1, b_2, b_3, …, c_1, c_2, c_3, …, where, for example, a_1, a_2, a_3, etc., are different from a.

(2) Logical operators: $\wedge$, $\vee$, $\oplus$, $\rightarrow$, whose names and interpretations are as follows:

Symbol	Name and Interpretation
$\wedge$	*and*
$\vee$	*or*
$\oplus$	*exclusive or (xor)*
$\rightarrow$	*sequence*

(3) Parentheses: ( ), which are used for punctuation.

These three sets of symbols constitute the *vocabulary* of the language of process logic. ■

or implies "at least one" and is the *inclusive* sense of or. *xor* implies "exactly one" and is the *exclusive* sense of or. In propositional logic, the inclusive sense of or is standard, whereas in process phenomena, the exclusive sense of or is quite common. Although in propositional logic, a proposition involving *xor* can be equivalently expressed by a proposition using *or*, *and*, and *not* (note that *not* is not included in process logic), this is not a case in process logic. Hence, in addition to *or*, *xor* is a necessary logical operator in process logic.

Definition 4 (formula). A *formula* of the language of process logic is any sequence of elements of the vocabulary of the language of process logic. ■

Definition 5 (well-formed formula). The *well-formed formulas* (or simply *wffs*) of the language of process logic are defined inductively by three *formation rules* that constitute the grammar of the language of process logic:

R1: Each process variable is a wff.
R2: If α, β, α_1, α_2, …, and α_n are wffs, then so are $(\alpha_1, \alpha_2, \ldots, \alpha_n)\wedge$, $\wedge(\alpha_1, \alpha_2, \ldots, \alpha_n)$, $(\alpha_1, \alpha_2, \ldots, \alpha_n)\vee$, $\vee(\alpha_1, \alpha_2, \ldots, \alpha_n)$, $(\alpha_1, \alpha_2, \ldots, \alpha_n)\oplus$, $\oplus(\alpha_1, \alpha_2, \ldots, \alpha_n)$, $(\alpha \rightarrow \beta)$.
R3: Every wff is obtained by a finite number of applications of *R1* and *R2*. ■

Table 1 summarizes wffs that are built up from simpler wffs using logical operators.

Table 1. Wffs That Are Built Up from Simpler Wffs

Name	wff	Graphical Structure
and-join	$(\alpha_1, \alpha_2, \ldots, \alpha_n)\wedge$	Figure 2(a)
and-split	$\wedge(\alpha_1, \alpha_2, \ldots, \alpha_n)$	Figure 2(b)
xor-join	$(\alpha_1, \alpha_2, \ldots, \alpha_n)\oplus$	Figure 2(c)
xor-split	$\oplus(\alpha_1, \alpha_2, \ldots, \alpha_n)$	Figure 2(d)
or-join	$(\alpha_1, \alpha_2, \ldots, \alpha_n)\vee$	Figure 2(e)
or-split	$\vee(\alpha_1, \alpha_2, \ldots, \alpha_n)$	Figure 2(f)
sequence	$\alpha \rightarrow \beta$	Figure 2(g)

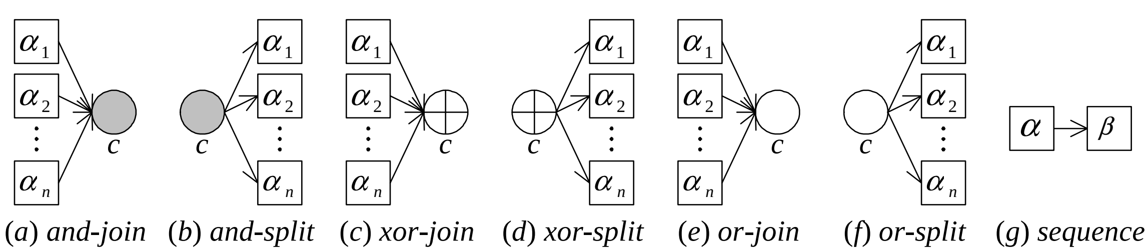

Figure 2. Structures in Standard Process Graphs

Definition 6 (process argument form). For a standard process graph P_S, the corresponding *process argument form F* of the language of process logic is written as

$$s, P_1, P_2, \ldots, P_n \vdash e$$

where
(1) *s* and *e* are the start and end vertices in P_S, and represent two process variables in process logic,
(2) $P_1, P_2, \ldots, P_n$ constitute a unique, finite set of *sequence* wffs,

(3) s, P_1, P_2, …, P_n constitute a finite set of premises, separated by commas, of F,
(4) e is the conclusion of F,
(5) premises are intended to prove the conclusion, and
(6) the symbol $\vdash$ is called an assertion sign. ■

For instance, the entire process graph in Figure 1 can be expressed as seven wffs as follows:

(1) s
(2) $s \rightarrow c_1$
(3) $c_1 \rightarrow \wedge(a_1, a_2)$
(4) $a_2 \rightarrow a_3$
(5) $(a_1, a_3)\wedge\rightarrow c_2$
(6) $c_2 \rightarrow e$
$\therefore$ (7) e

These wffs can be written in a process argument form:

$$s, s \rightarrow c_1, c_1 \rightarrow \wedge(a_1, a_2), a_2 \rightarrow a_3, (a_1, a_3)\wedge \rightarrow c_2, c_2 \rightarrow e \vdash e$$

Conversion of Standard Process Graphs into Wffs

According to the conversion rules in Table 2, each *sequence* process construct can be converted into a unique *sequence* wff, and each process construct involving a control vertex can be converted into two unique *sequence* wffs. As a result, a standard process graph P_S can be converted into a unique, finite set of *sequence* wffs.

Table 2. Rules of Converting Process Constructs into Wffs

Process Construct	Graphical Representation	Wff
Sequence	Figure 3(a)	$v \rightarrow w$
AND	Figure 3(b)	$(v_1, v_2, \ldots, v_m)\wedge \rightarrow c$, $c \rightarrow \wedge(w_1, w_2, \ldots, w_n)$
XOR	Figure 3(c)	$(v_1, v_2, \ldots, v_m)\oplus \rightarrow c$, $c \rightarrow \oplus(w_1, w_2, \ldots, w_n)$
OR	Figure 3(d)	$(v_1, v_2, \ldots, v_m)\vee \rightarrow c$, $c \rightarrow \vee(w_1, w_2, \ldots, w_n)$

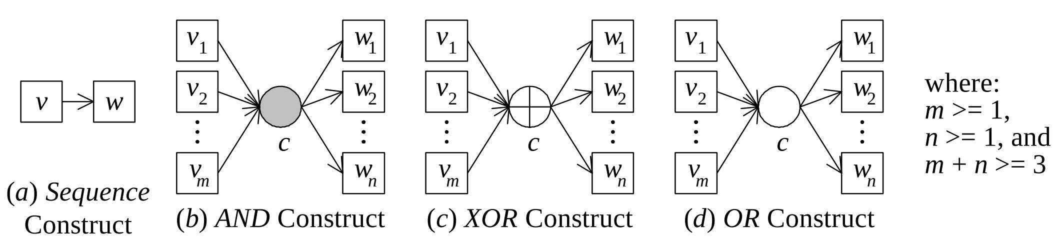

Figure 3. Four Process Constructs
(See Appendix A)

Semantics of Process Logic

In this section, we define the semantics of process logic by means of truth values.

Semantics of Logical Operators

For an *and-join* structure in Figure 2(a), a control vertex *c* is activated after all activity vertices $\alpha_1, \alpha_2, \ldots \alpha_n$ are executed. *C* is not activated when at least one of $\alpha_1, \alpha_2, \ldots \alpha_n$ is not executed. Generally, an *and-join* wff is true when all of its components are true.

Definition 7 (*and-join*). The truth value of an *and-join* wff is defined as

$$(\alpha_1, \alpha_2, \ldots, \alpha_n)\wedge = \begin{cases} 1, \text{if } \alpha_1 = \alpha_2 = \ldots = \alpha_n = 1 \\ 0, \text{otherwise} \end{cases} \quad (n \geq 2).$$ ■

For an *and-split* structure in Figure 2(*b*), after a control vertex *c* is activated, all activity vertices $\alpha_1, \alpha_2, \ldots, \alpha_n$ are executed. If *c* is not activated, then none of $\alpha_1, \alpha_2, \ldots, \alpha_n$ is executed. In any case, it is impossible that only some but not all of $\alpha_1, \alpha_2, \ldots, \alpha_n$ are executed. Generally, if an *and-split* wff is true, then all of its components are true.

Definition 8 (*and-split*). The truth values of the components of an *and-split* wff are defined as

$$\alpha_1 = \alpha_2 = \ldots = \alpha_n = \begin{cases} 0, \text{if } \wedge(\alpha_1, \alpha_2, \ldots, \alpha_n) = 0 \\ 1, \text{if } \wedge(\alpha_1, \alpha_2, \ldots, \alpha_n) = 1 \end{cases} \quad (n \geq 2).$$ ■

For an *xor-join* structure in Figure 2(*c*), a control vertex *c* is activated after exactly one of activity vertices $\alpha_1, \alpha_2, \ldots, \alpha_n$ is executed; otherwise, *c* is not activated. Generally, an *xor-join* wff is true when exactly one of its components is true.

Definition 9 (*xor-join*). The truth value of an *xor-join* wff is defined as

$$(\alpha_1, \alpha_2, \ldots, \alpha_n)\oplus = \begin{cases} 1, \text{if } \alpha_i = 1\,(1 \leq i \leq n) \text{ and } \alpha_j = 0\,(j \in \{1, 2, \ldots, n\} - i) \\ 0, \text{otherwise} \end{cases} \quad (n \geq 2).$$ ■

For an *xor-split* structure in Figure 2(*d*), after a control vertex *c* is activated, exactly one of activity vertices $\alpha_1, \alpha_2, \ldots, \alpha_n$ is executed. If *c* is not activated, then none of $\alpha_1, \alpha_2, \ldots, \alpha_n$ is executed. In any case, it is impossible that more than one of $\alpha_1, \alpha_2, \ldots, \alpha_n$ is executed. Generally, if an *xor-split* wff is true, then exactly one of its components is true.

Definition 10 (*xor-split*). The truth values of the components of an *xor-split* wff are defined as

$$\text{If } \oplus(\alpha_1, \alpha_2, \ldots, \alpha_n) = 0, \text{ then } \alpha_1 = \alpha_2 = \ldots = \alpha_n = 0 \; (n \geq 2).$$
$$\text{If } \oplus(\alpha_1, \alpha_2, \ldots, \alpha_n) = 1, \text{ then } \alpha_i = 1 \; (1 \leq i \leq n) \text{ and } \alpha_j = 0 \; (j \in \{1, 2, \ldots, n\} - i) \; (n \geq 2).$$ ■

For an *or-join* structure in Figure 2(*e*), a control vertex *c* is activated after at least one of activity vertices $\alpha_1, \alpha_2, \ldots, \alpha_n$ is executed. *c* is not activated when none of $\alpha_1, \alpha_2, \ldots, \alpha_n$ is executed. Generally, an *or-join* wff is true when at least one of its components is true.

Definition 11 (*or-join*). The truth value of an *or-join* wff is defined as

$$(\alpha_1, \alpha_2, \ldots, \alpha_n)\vee = \begin{cases} 0, \text{if } \alpha_1 = \alpha_2 = \ldots = \alpha_n = 0 \\ 1, \text{otherwise} \end{cases} \quad (n \geq 2).$$ ■

For an *or-split* structure in Figure 2(*f*), after a control vertex *c* is activated, at least one of activity vertices $\alpha_1, \alpha_2, \ldots, \alpha_n$ is executed. If *c* is not activated, then none of $\alpha_1, \alpha_2, \ldots, \alpha_n$ is executed. Generally, if an *or-split* wff is true, then at least one of its components is true.

Definition 12 (*or-split*). The truth values of the components of an *or-split* wff are defined as

$$\text{If } \vee(\alpha_1, \alpha_2, \ldots, \alpha_n) = 0, \text{ then } \alpha_1 = \alpha_2 = \ldots = \alpha_n = 0 \ (n \geq 2).$$
$$\text{If } \vee(\alpha_1, \alpha_2, \ldots, \alpha_n) = 1, \text{ then } \exists\ \alpha_i \ (1 \leq i \leq n),\ \alpha_i = 1 \ (n \geq 2).$$

■

For a *sequence* structure in Figure 2(*g*), if α and β are both activity vertices, after α is executed, β will definitely be executed. If α is not executed, then β will not be executed. In any case, it is impossible that α is executed but β is not executed, or that α is not executed but β is executed. Generally, the antecedent and consequent of a *sequence* wff always have the same truth value.

Definition 13 (*sequence*). The truth value of the consequent of a *sequence* wff $\alpha \rightarrow \beta$ is defined as

$$\beta = \begin{cases} 0, \text{if } \alpha = 0 \\ 1, \text{if } \alpha = 1 \end{cases}.$$

■

The semantics of logical operators can also be expressed using truth tables.

Definition 14 (truth table). A *truth table* is a tabular description of all feasible truth value assignments involved in wffs and is used to display calculations of truth values. ■

Table 3 gives the truth table of *sequence*. As a distinctive feature in process logic, the truth value of a *sequence* $\alpha \rightarrow \beta$ itself is of no interest. α and β are either both true or both false, thus always giving a "true" truth value to a *sequence* $\alpha \rightarrow \beta$ in the sense of propositional logic.

In a truth table, on the left-hand side of double vertical lines are all possible truth value assignments that are used for calculation, and on the right-hand side are the calculation results on the basis of the logical operator involved and the truth values on the left-hand side. Given bivalence, it is possible that a truth table completely describes the truth values in every feasible situation for a wff.

Table 3. Truth Table of *sequence* $\alpha \rightarrow \beta$

α	β
0	0
1	1

Table 4 through Table 7 show the truth tables of *and*, *xor*, and *or* wffs, each of which contains three components. Truth tables of *and-join*, *xor-join*, and *or-join* that contains n ($n \geq 2$) components comprise 2^n rows, excluding the heading rows that are not an official part of a truth table.

Table 4. Truth Table of *and-join, xor-join,* and *or-join* with Three Components

			Figure 4(a)	Figure 4(c)	Figure 4(e)
			and-join	*xor-join*	*or-join*
α_1	α_2	α_3	$(\alpha_1, \alpha_2, \alpha_3)\wedge$	$(\alpha_1, \alpha_2, \alpha_3)\oplus$	$(\alpha_1, \alpha_2, \alpha_3)\vee$
0	0	0	0	0	0
1	0	0	0	1	1
0	1	0	0	1	1
0	0	1	0	1	1
1	1	0	0	0	1
1	0	1	0	0	1
0	1	1	0	0	1
1	1	1	1	0	1

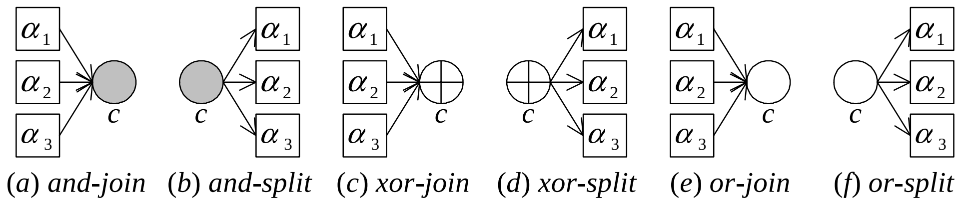

Figure 4. *and, xor,* and *or* Structures with Three Activity Vertices

Table 5. Truth Table of *and-split* with Three Components

Figure 4(b) *and-split* $\wedge(\alpha_1, \alpha_2, \alpha_3)$	α_1	α_2	α_3
0	0	0	0
1	1	1	1

Table 6. Truth Table of *or-split* with Three Components

Figure 4(f) *or-split* $\vee(\alpha_1, \alpha_2, \alpha_3)$	α_1	α_2	α_3
0	0	0	0
1	1	0	0
1	0	1	0
1	0	0	1
1	1	1	0
1	1	0	1
1	0	1	1
1	1	1	1

Table 7. Truth Table of *xor-split* with Three Components

Figure 4(d) *xor-split* $\oplus(\alpha_1, \alpha_2, \alpha_3)$	α_1	α_2	α_3
0	0	0	0
1	1	0	0
1	0	1	0
1	0	0	1

Truth Tables for Process Argument Forms

Truth tables can also provide a rigorous evaluation of deductive validity of process argument forms.

Definition 15 (*valid process argument form*). A process argument form F is *valid* if its conclusion e is true when its premise s is true, if no other components involved in the same consequent as e in a *sequence* wff are true when e is true, and if all premises and all process variables are used in proving the conclusion e. ■

Truth tables are actually exhaustive lists of all possible situations. Thus, we can use truth tables to verify whether a process argument form is valid.

We illustrate with an example the procedure of verifying the validity of a process argument form without giving the formal algorithm due to the space limits.

For example, a standard process graph P_S in Figure 1 can be converted into a unique process argument form F: $s, s \rightarrow c_1, c_1 \rightarrow \wedge(a_1, a_2), a_2 \rightarrow a_3, (a_1, a_3)\wedge\!\rightarrow c_2, c_2 \rightarrow e \vdash e$. F can be rewritten to show the number of each premise: (1) s, (2) $s \rightarrow c_1$, (3) $c_1 \rightarrow \wedge(a_1, a_2)$, (4) $a_2 \rightarrow a_3$, (5) $(a_1, a_3)\wedge \rightarrow c_2$, (6) $c_2 \rightarrow e \vdash e$.

Table 8 displays the verification of the validity of F, with the top heading row showing the number of each premise involved in the truth value calculation. In a standard process graph, the start vertex s is executed unconditionally when the process starts. Hence, at the beginning of the calculation, only s is set to 1. The truth values of all other wffs must be calculated. The truth value of c_1 in premise (2) is determined on the basis of s, and then the truth value of $\wedge(a_1, a_2)$ in premise (3) is determined on the basis of c_1, and so on. Finally, we obtain the truth value of the conclusion e, which is 1. Therefore, it can be concluded that F is valid, because e is 1 when s is 1, and all premises and all process variables are used in proving the conclusion e.

Table 8. Verification of the Validity Process Argument Form F Corresponding to P_S in Figure 1

(1)	(2)	(3)	(3)		(4)	(5)	(5)	(6)
s	c_1	$\wedge(a_1, a_2)$	a_1	a_2	a_3	$(a_1, a_3)\wedge$	c_2	e
1	1	1	1	1	1	1	1	1

Applying Process Logic to Verifying Process Models

In this section, we use process logic to verify process models and detect process anomalies.

Verification of the Correctness of Process Graphs

If the corresponding process argument form F of a standard process graph P_s is valid, then P_s is correct. Because truth tables exhaustively list all possible situations for a process argument form, we can use truth tables to determine whether a process argument form is valid and thus the underlying process model is correct.

Detection of Process Anomalies

Process logic can be used to detect process anomalies such as deadlock. A *deadlock* refers to a situation in which a process instance gets into a stalemate such that no activity can be further executed (Verbeek et al. 2001). As shown in Figure 5, when an *AND-Join* vertex c_2 is mismatched with an *XOR-Split* vertex c_1, a deadlock occurs at c_2. The process argument form F corresponding to the standard process graph P_S in Figure 5 is s, $s \rightarrow c_1$, $c_1 \rightarrow \oplus(a_1, a_2)$, $(a_1, a_2)\wedge \rightarrow c_2$, $c_2 \rightarrow e \vdash e$, for which truth tables are constructed in Tables 9 and 10.

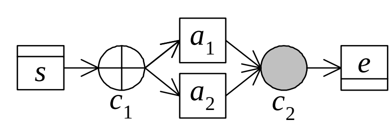

Figure 5. An Example of Deadlock

Because e is 0 when s is 1, F is invalid and thus P_S is incorrect. The problem is that $(a_1, a_2)\wedge$ is always 0 when s is 1; in other words, the *AND-Join* vertex c_2 can never be activated, thus preventing the process from completion.

Table 9. The First Truth Table

s	c_1	$\oplus(a_1, a_2)$
1	1	1

Table 10. The Second Truth Table

$\oplus(a_1, a_2)$	a_1	a_2	$(a_1, a_2)\wedge$	c_2	e
1	1	0	0	0	0
1	0	1	0	0	0

Conclusions

In this paper, we proposed process logic as a logical formalism and mathematical tool to verify the correctness of process models. Although process logic borrows a great amount from propositional logic, process logic is defined to specifically describe process phenomena such as join and split structures in process models. Many such process phenomena cannot be properly represented by using propositional logic.

We demonstrated that process logic is capable of verifying the correctness of an arbitrary activity-based process model by converting a standard process graph into a process argument form, and then determining the validity of the process argument form through truth tables. Truth tables constitute a systematic and exhaustive approach to process verification.

References

Bi, H. H., and Zhao, J. L. "Mending the Lag Between Commercial Needs and Research Prototypes: A Logic-Based Workflow Verification Approach," in *Proceedings of the 8th INFORMS Computing Society Conference*, H. K. Bhargava and N. Ye (Eds.), Chandler, AZ, January 8-10, 2003, pp. 191-212.

Bi, H. H., and Zhao, J. L. "Applying Propositional Logic to Workflow Verification," *Information Technology and Management* (5:3-4), 2004a, pp. 293-318.

Bi, H. H., and Zhao, J. L. "Process Graphs: A Formal Language for Business Process Modeling and Analysis," Working paper, Department of Supply Chain and Information Systems, Pennsylvania State University, 2004b.

Bonner, A. J., and Kifer, M. "An Overview of Transaction Logic," *Theoretical Computer Science* (133:2), 1994, pp. 205-265.

Choi, Y., and Zhao, J. L. "Handling Cycles in Workflow Verification by Feedback Identification and Partition," in *Proceedings of the 2003 International Conference on Information and Knowledge Engineering*, Las Vegas, Nevada, June 23-26, 2003.

Choi, Y., and Zhao, J. L. "Matrix-Based Abstraction and Verification for e-Business Processes," in *Proceedings of the 1st Workshop on e-Business*, Barcelona, Spain, December 14-15, 2002, pp. 154-165.

Davenport, T. H. *Process Innovation: Reengineering Work through Information Technology*, Harvard Business School Press, Boston, MA, 1993.

Kifer, M. "Transaction Logic for the Busy Workflow Professional," unpublished manuscript, 1996 (available online at CiteSeer.IST: **http://citeseer.ist.psu/edu/**).

Lin, F. R., Yang, M. C., and Pai, Y. H. "A Generic Structure for Business Process Modeling," *Business Process Management Journal* (8:1), 2002, pp. 19-41.

Mayer, R. J., Menzel, C. P., Painter, M. K., deWitte, P. S., Blinn, T., and Perakath, B. *Information Integration for Concurrent Engineering: IDEF3 Process Description Capture Method Report*, Knowledge Based Systems, Incorporated, College Station, TX, 1995.

Sadiq, W., and Orlowska, M.E. "Analyzing Process Models Using Graph Reduction Techniques," *Information Systems* (25:2), 2000, pp. 117-134.

Van der Aalst, W. M. P., Hirnschall, A., and Verbeek, H. M. W. "An Alternative Way to Analyze Workflow Graphs," in *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE'02)*, Springer-Verlag, Berlin, 2002, pp. 535-552.

Van der Aalst, W. M. P., and Kumar, A. "XML-Based Schema Definition for Support of Inter-organizational Workflow, *Information Systems Research.* (14:1), 2003, pp. 23-46.

Verbeek, H. M. W., Basten, T., and Van der Aalst, W. M. P. "Diagnosing Workflow Processes Using Woflan," *The Computer Journal* (44:4), 2001, pp. 246-279.

Appendix A. Process Graphs

In this appendix, we give the key definitions of process graphs (Bi and Zhao 2004b), where $d^-(v)$ is in-degree of v, $d^+(v)$ is out-degree of v, $d(v)$ is degree of v, $N^-(v)$ is in-neighborhood of v, $N^+(v)$ is out-neighborhood of v, and $N(v)$ is neighborhood of v.

Definition 16 (*process graph*). A *process graph* is a 5-tuple $P = (V_A(P), V_C(P), A(P), s, e)$, where

(1) *activity vertex set* $V_A(P)$ is a non-empty finite set of distinct elements called *activity vertices*, and $\forall\, v \in V_A(P)$, $d^-(v) = d^+(v) = 1$,

(2) *control vertex set* $V_C(P)$ is a finite set of distinct elements called *control vertices*, and $\forall\, v \in V_C(P)$, $d^-(v) \geq 1$, $d^+(v) \geq 1$, $d(v) \geq 3$, and $T_C(v) \in \{AND, XOR, OR\}$ is the type of v,

(3) s is the *start vertex*, and $d^-(s) = 0$, $d^+(s) = 1$,

(4) e is the *end vertex*, and $d^-(e) = 1$, $d^+(e) = 0$,

(5) *directed arc set* (or simply *arc set*) $A(P)$ is a non-empty finite set of distinct ordered pairs called *arcs* of distinct elements of *vertex set* $V(P) = V_A(P) \cup V_C(P) \cup \{s, e\}$,

(6) P is connected, and

(7) $V_A(P) \cap V_C(P) = \emptyset$, $V_A(P) \cap \{s, e\} = \emptyset$, and $V_C(P) \cap \{s, e\} = \emptyset$. ■

Figure A1 gives the graphical notation for process graphs.

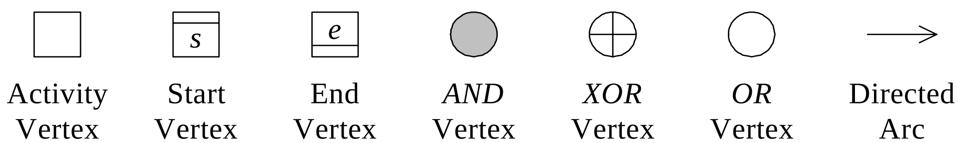

Figure A1. Graphical Notation for Process Graphs

Definition 17 (*standard process graph*). A *standard process graph* P_S is a process graph P in which for any control vertex $c \in V_C(P)$, $N(c) \subseteq V_A(P) \cup \{s, e\}$. A process graph that is not a standard process graph is called a *non-standard process graph*. ■

Definition 18 (*process construct*). In a standard process graph P_S, a *process construct* (or simply *construct*) consists of (1) u, $v \in V_A(P_S) \cup \{s, e\}$, and uv or $vu \in A(P_S)$, or (2) $c \in V_C(P_S)$, $N(c)$, vc, and cw, $\forall\, v \in N^-(c)$ and $\forall\, w \in N^+(c)$. ■

Figure A2 illustrates four process constructs. A *sequence construct* consists of two non-control vertices joined by an arc. An *AND* (*XOR*, *OR*) *construct* consists of an *AND* (*XOR*, *OR*) vertex with its neighbors and arcs with which the *AND* (*XOR*, *OR*) vertex and its neighbors are incident. In a *sequence* construct, the execution of a vertex causes the execution of the other vertex. In an *AND* (*XOR*, *OR*) construct, the *AND* (*XOR*, *OR*) vertex is activated by the execution of all (exactly one, at least one) of its in-neighbors, and then the activated control vertex, in turn, causes the execution of all (exactly one, at least one) of its out-neighbors.

Definition 19 (*correct process graph*). A process graph P is *correct* if every instance walk of P with s as the initial vertex can be extended to an (s, e)-instance-walk with e being executed exactly once, and if every activity vertex in P is in at least one (s, e)-instance-walk. Otherwise, P is an *incorrect process graph*. ■

An instance walk is simply a process instance expressed as a walk in a graph.

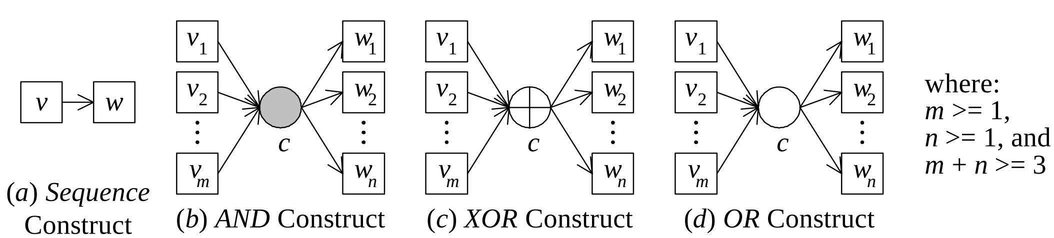

Figure A2. Four Process Constructs