

Automatic workflow verification and generation

Shiyong Lu^{a,*}, Arthur Bernstein^b, Philip Lewis^b

^aDepartment of Computer Science, Wayne State University, Detroit, MI 48202, USA

^bDepartment of Computer Science, State University of New York, Stony Brook, NY 11790, USA

Received 12 August 2003; received in revised form 11 April 2004; accepted 7 October 2005

Communicated by H. Katsuno

Abstract

Correctness is an important aspect of workflow management systems. However, most of the workflow literature focuses only on the modeling aspects and assumes that a workflow is correct if during the execution it respects the control and data dependency specified by the workflow designer. To address the correctness question properly we propose a new workflow model based on Hoare semantics that allows to: (1) automatically check if the desired outcome of a workflow can be produced by its actual implementation, (2) automatically synthesize a workflow implementation from the workflow specification and a given task library.

In particular we: (1) formalize the semantics of workflows and tasks with pre- and postconditions, (2) for each control construct we provide a set of sound inference rules formalizing its semantics. While most of our workflow constructs are standard, two of them are new: the universal and the existential constructs. We then describe algorithms for automatically checking the correctness of workflows and for automatic workflow generation.

© 2005 Elsevier B.V. All rights reserved.

Keywords: Workflow verification; Workflow generation; Workflow correctness; Web services; Hoare logic

1. Introduction

Correctness is an important aspect of workflow management systems. Although most of the workflow community is interested in workflow modeling aspects, a few researchers have been investigating techniques for supporting workflow correctness [7,18,45,60]. An excellent overview of correctness issues in workflow management is given in [42]. More recently, a formalization of workflows based on set and graph theory that addresses correctness issues is given in [6].

Some researchers focused on the correctness aspects that ensure data consistency when concurrency and failures are present. These techniques emerge from the areas of extended transaction models [3,32,30,31,67], multi-databases [47,19] and transactional workflows [60]. However, the constraints that the data and control flow have to satisfy were not discussed in a formal way. Other researchers focused on data and control flow requirements. These techniques include control flow graphs, triggers (i.e., event-condition-action rules) [27], temporal constraints [26,62] and net-based approaches [1]. However, most of these approaches, although formal, assume the workflow is correct if the constraints on data and control flow are satisfied during execution. Whether, the final state of the whole workflow is

* Corresponding author.

E-mail addresses: shiyong@cs.wayne.edu (S. Lu), art@cs.sunysb.edu (A. Bernstein), pml@cs.sunysb.edu (P. Lewis).

the desired one is neither specified nor proved. Thus, to the best of our knowledge, a formal framework for reasoning about the correctness of a workflow with respect to a specification of its outcome is still missing.

Recently, a semantic-correctness theory has been developed for transaction processing systems [14,11,12]. This theory is useful both for improving the performance of a transaction processing system [13] and for reasoning about execution correctness of transactions running at different isolation levels [15]. In this paper we extend this semantic-correctness theory to modeling and reasoning about workflows. Specifically, we develop a formal model in which workflows are assumed to be constructed from a library of tasks to promote task reuse. The semantics of tasks and workflows is specified in terms of pre- and postconditions, and a sound inference rule is provided to precisely specify each of our workflow constructs. Based on this model we develop algorithms that automatically: (1) check if a workflow implementation satisfies its specification, (2) synthesize a workflow implementation from the workflow description and a given task-library. Since the verification and synthesis problems are in general undecidable, we restrict our attention to finite data domains only.

The rest of the paper is organized as follows: Section 2 introduces the main features of our model using a negotiation workflow example. Section 3 defines some preliminary concepts. Section 4 presents the task model. Section 4 describes the workflow model. Section 6 formalizes the semantics of workflow constructs which are used to build a workflow from tasks or from other workflows. Section 7 describes an algorithm that verifies the correctness of a given workflow. Section 8 describes an algorithm that automatically generates a correct workflow if the algorithm can find it, based on the workflow description and a given task library. Section 9 describes some related work and provides some discussion; finally, Section 10 concludes the paper and points out some remaining future work.

2. Workflow model features

A workflow is generally used to model a complex business process. It can be viewed as a directed graph, in which the nodes represent tasks and the arcs describe their execution ordering. The execution of tasks in a workflow is controlled by a workflow controller. The state of the workflow is maintained in the controller's database.

Example 2.1. To introduce our workflow notation and model we use a trade negotiation example. Part of the negotiation workflow is shown in Fig. 1. The workflow describes a negotiation between parties A and B . It starts with the execution of task $Start(A, B)$, in which A establishes a connection to B and makes an initial bid. During negotiation, either party can choose to accept an offer or provide a counter offer by executing task $Neg(x)$. If one party chooses to accept an offer, the other party will agree with the acceptance by executing task $Agree(x)$.¹ Thereafter, both parties enter the *settlement stage*. Based on the number of brokerage branches involved in the negotiation, a result obtained by executing task $BrokerageHouse$, either the branches corresponding to both parties settle the deal by executing $Settle(x)$ for all $x \in \{A, B\}$ (the right box), or if both parties are from the same branch, that branch settles the deal by executing $Settle(x)$ for some $x \in \{A, B\}$ (the left box).

This example illustrates the following features of our workflow model:

1. A task may be parameterized. For example, task $Neg(x)$ is parameterized by the participating party $x \in \{A, B\}$.
2. A task T might take one of several actions non-deterministically. Here, the non-determinism means that the workflow controller has no control over which action T might take during execution. One situation is that the user decides which action of T to take. For example, the result of task $Neg(x)$ is either $newbid(x)$ (the party provides a new bid) or $agree(x)$ (the party agrees with the other party's proposal).
3. A task may assign a value to a variable accessible to following tasks. For example, task $BrokerageHouse$ will write variable br containing the number of parties involved in the negotiation.
4. A branch traversal may be guarded with a condition. For example, $br = 1$ and $br = 2$ guard the outgoing left and right branches of the \oplus connector.
5. An existential workflow node $\oplus_{x \in D} W(x)$ (to be shown as a special case of a conditional workflow) non-deterministically runs workflow $W(x)$ for only one instance $x \in D$.
6. A universal workflow node $\otimes_{x \in D} W(x)$ concurrently runs workflow $W(x)$ for all instances $x \in D$.

¹ A real trade negotiation workflow would allow both parties to withdraw from the negotiation before they enter the settlement stage.

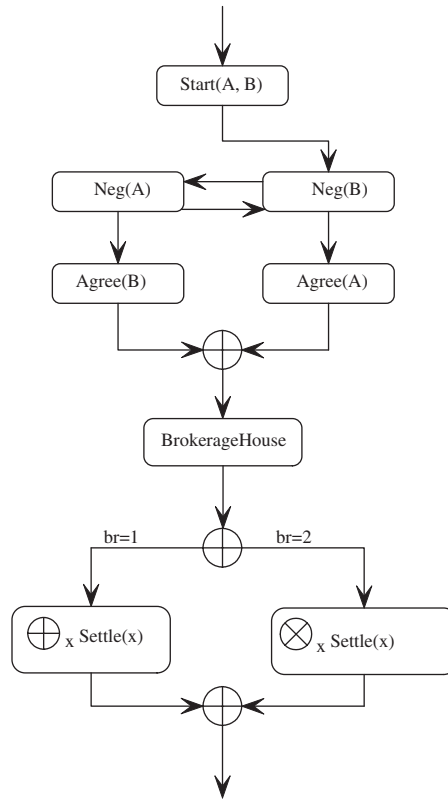


Fig. 1. A trade negotiation workflow example.

3. Preliminary definitions

A *predicate* is a statement that asserts a relationship about workflow objects. It has a unique identifier and a finite sequence of parameters. The value of each parameter is drawn from a finite domain. A predicate is called a *proposition* if its parameter sequence is empty or all its parameters are instantiated. For example, $\text{newbid}(x)$ represents the relation “there is a new bid proposed by x that has not been responded to.” The workflow database stores the value of each proposition of a workflow instance.

A *variable*, y , is a semantic entity that has an ordered finite domain denoted by $\text{Dom}(y)$. Its value is stored in the workflow database and is accessible to all tasks. For each variable y , predicate $\text{gen}(y)$ states that “ y ’s value is defined and available”. For example, br represents “the number of branches involved in the negotiation” and the execution of task `BrokerageHouse` makes $\text{gen}(br)$ true. The workflow database stores the value of each variable.

A *condition* is a comparison between a variable y and a constant from y ’s domain. The interpretation of a comparison operator is defined as part of the model, and a condition on y evaluates to true if y has a value ($\text{gen}(y)$ is true) and the comparison is satisfied, and false otherwise. For example, for a variable y whose domain is integer, the following comparisons are defined: $<$, \leq , $=$, $>$, \geq . Although the model supports variables of different ordered domains, in this paper, we assume each variable has an integer domain and only the above five comparisons are considered.

An *atomic formula* is a predicate or a condition. To describe the expected outcome of a workflow, we use the *well-formed formulas* of the first order logic defined as follows [40]:

Definition 3.1 (Well-formed formula (wff)). A *well-formed formula* (wff) is defined recursively as follows:

1. An atomic formula is a wff.
2. If α and β are wffs, then $\neg\alpha$, $\alpha \wedge \beta$ and $\alpha \vee \beta$ are wffs.
3. If ψ is a wff, and x is a variable, then $\forall_x \psi$ and $\exists_x \psi$ are wffs.
4. wffs are generated only by a finite number of applications of rule 1, 2, 3.

A *state* is an assignment of values to variables and propositions. Each wff denotes a set of states. For example, $\neg a \wedge b \wedge (x > 5)$ represents a set of states in which “ a is false, b is true, and x has a value that is greater than 5”. Given a wff A and a state σ , the set of states represented by A is denoted by $\llbracket A \rrbracket$, and σ *satisfies* A if and only if $\sigma \in \llbracket A \rrbracket$. To specify the effect of executing a task, we introduce the notion of action as follows.

Definition 3.2 (Action). A *literal* is a predicate or a condition (*positive literal*), or its negation (*negative literal*). An *action* is a list of literals. Given an action A , the sets of positive and negative literals in A are represented by A^+ and A^- , respectively. For convenience, in this paper, we denote an action as a conjunction of literals.

Semantically, each action specifies the changes that a task makes to the state by assignments. For example, action $a \wedge \neg b$ represents “the execution assigns true to a and false to b ”.

4. The task model

Tasks are the basic building blocks of workflows. We assume that each task T has the ACID properties of conventional transactions [34] and it must be initiated in a state satisfying some wff called its *precondition*, to ensure its execution correctness. During execution, a task T takes one of its actions non-deterministically (the non-determinism feature of a task is illustrated in Example 2.1). A formal definition of task specification is as follows.

Definition 4.1 (Task specification). A task specification is a triple

$$\{P(\bar{x})\} T(\bar{x}) [Q_1(\bar{x}), Q_2(\bar{x}), \dots, Q_n(\bar{x})], \quad (1)$$

where \bar{x} is a sequence of parameters. $P(\bar{x})$ is a wff called T 's *precondition*, and each $Q_i(\bar{x})$ is an action of T .

Example 4.2. Task `Settle(x)` in Example 2.1 can be specified by

$$\{da\} \text{Settle}(x) [stl(x)],$$

where da is a proposition which asserts that “the deal is agreed to by both parties” and $stl(x)$ asserts that “party x has settled the deal”.

In the remainder of the paper, the parameter sequence \bar{x} is omitted when it is implicit. Pictorially, a task T is represented by a box labeled with T 's identifier having one input edge and n output edges, as shown in Fig. 2. The input edge is labeled with T 's precondition P and each of T 's output edges is labeled with Q_i ($i = 1, 2, \dots, n$).

Definition 4.3 (Frame semantics). We define the following so called *frame semantics* for task specification $\{P\} T [Q_1, Q_2, \dots, Q_n]$: if T is initiated in a state S that satisfies wff P' and $P' \Rightarrow P$, then when T terminates, the resulting state satisfies $result(P', Q_i)$, where Q_i is chosen by T *non-deterministically*² and the function $result$ is defined by $result(P', Q_i) = (P'/Q_i) \wedge Q_i$. Here, (P'/Q_i) represents wff P' with every literal l deleted if $l \in Q_i$ or $\neg l \in Q_i$.

For example, given $P' = a \wedge b \wedge \neg c$, and $Q_i = \neg b \wedge c \wedge d$, we have $(P'/Q_i) = a$ and $result(P', Q_i) = a \wedge \neg b \wedge c \wedge d$. The $result$ function formalizes the following:

1. For each literal l such that $l \in Q_i^+$, its value becomes true.
2. For each literal l such that $\neg l \in Q_i^-$, its value becomes false.
3. For each literal in $l \in P'$ such that $l, \neg l \notin Q_i$, its value remains unchanged.

Example 4.4. Suppose task T is specified by $\{a \wedge b\} T [\neg a \wedge c, \neg b \wedge d]$ and is initiated in a state satisfying $a \wedge b \wedge \neg c$. Then when T terminates, the resulting state satisfies $result(a \wedge b \wedge \neg c, \neg a \wedge c) = \neg a \wedge b \wedge c$ for the first action or $result(a \wedge b \wedge \neg c, \neg b \wedge d) = a \wedge \neg b \wedge \neg c \wedge d$ for the second action.

² One possible implementation is, an integer returned by the invocation of T indicates which action of T has been taken, for example i indicates action Q_i has been taken.

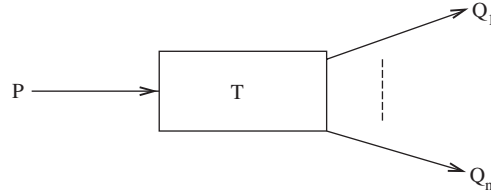


Fig. 2. The task model.

One might be tempted to specify a task $\{a\} T [b, c]$ by $\{a\} T [b \vee c]$ so that each task has one input edge and one output edge that corresponds to one-entry–one-exit structured notion of programming languages. However, by doing so, we would lose information since the execution of $\{a\} T [b \vee c]$ contains one more possibility which is not implied by the first specification: assign true to both b and c . Using exclusive-or between b and c will not do it either since if b is true initially, then the execution of T might terminate in a state in which both b and c are true. How to represent a task as an one-input–one-output unit and how to generate workflow automatically based on this new task model is an interesting future work. This paper presents our workflow verification and generation algorithms based on the task model defined in this section.

A set of tasks is used to build a task library and they can be reused to build different workflows with the workflow constructs defined in Section 6.

Definition 4.5 (Task library). A task library is a tuple $(\mathcal{P}, \mathcal{V}, \mathcal{C}, \mathcal{L}, \mathcal{T})$ where

1. \mathcal{P} is a finite set of predicates.
2. \mathcal{V} is a finite set of variables with finite ordered domains.
3. \mathcal{C} is a finite set of conditions defined on \mathcal{V} .
4. \mathcal{L} is the set of literals defined by \mathcal{P} , \mathcal{V} , and \mathcal{C} .
5. \mathcal{T} is a set of task specifications with preconditions and actions over \mathcal{L} .

5. The workflow model

We consider a workflow as a business process that is constructed from tasks in the task library using the constructs defined in Section 6.

Definition 5.1 (Workflow description). A workflow description is a triple

$$\{P(\bar{x})\} W(\bar{x}) \{Q_1(\bar{x}), Q_2(\bar{x}), \dots, Q_n(\bar{x})\}, \quad (2)$$

where \bar{x} is a sequence of parameters. $P(\bar{x})$ is a wff called W 's *precondition*. $Q_1(\bar{x}), Q_2(\bar{x}), \dots, Q_n(\bar{x})$ is a set of wffs, called W 's *postcondition*. Each Q_i is called an *outcome* of W .

Pictorially, if we ignore its internal structure, a workflow W can be represented by a box labeled with its identifier. It has one input edge and n output edges (see Fig. 3). The input edge is labeled with W 's precondition P , and each of T 's output edges is labeled with its corresponding outcome Q_i ($i = 1, 2, \dots, n$).

Definition 5.2 (Assertional semantics). We define the following so called *assertional semantics* for workflow description $\{P\} W \{Q_1, Q_2, \dots, Q_n\}$: when W is initiated in a state satisfying P , then when it terminates, the resulting state satisfies one of the Q_i ($i = 1, \dots, n$).

Definition 5.3 (Correctness of a workflow). A workflow W is *correct* with respect to workflow precondition $WPre$ and workflow postcondition $WPost_1, \dots, WPost_n$ if $\{WPre\} W \{WPost_1, \dots, WPost_n\}$ is a theorem.

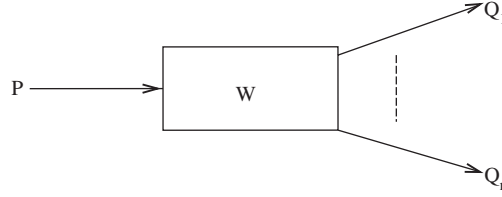


Fig. 3. An abstraction of a workflow.

The differences between frame and assertional semantics are summarized as follows:

- The frame semantics of a task specifies fully its functionality when it is executed from a state satisfying the task's precondition. In this way, a task can be reused in different workflows to implement different desirable assertional semantics.
- Assertional semantics does not fully specify the functionality of a workflow. For each triple (2), we might have several implementations from different tasks with different frame semantics (see Example 5.4).
- We can infer the assertional semantics of a task from its frame semantics, but not the other way around since for triple (2), there might exist several implementations.

Example 5.4. Given a workflow description $\{\neg a\} W \{a \vee b\}$, any task with the following frame semantics can be used to implement workflow W :

1. $\{\neg a\} T_1 [a]$.
2. $\{\neg a\} T_2 [b]$.
3. $\{\neg a\} T_3 [a \wedge b]$.
4. $\{\neg a\} T_4 [a, b]$.
5. $\{true\} T_5 [a]$.

Example 5.5. Suppose the frame semantics specification of task T is $\{\neg a\} T [a \wedge b]$ and workflow W is described by its assertional semantics $\{\neg a\} W \{a \wedge b\}$. Suppose the initial state satisfies $\neg a \wedge \neg b \wedge \neg c$. If we execute T , then after T terminates, the resulting state satisfies $a \wedge b \wedge \neg c$, whereas if we execute W , then after W terminates, the resulting state satisfies $a \wedge b$ and the value of c might be true or might be false.

The following inference rules, called *frame rules*, establish the relationship between the frame and assertional semantics of a workflow. In particular, one can infer from frame semantics to assertional semantics, but not the other way around.

$$\frac{\{P\} T [Q_1, Q_2, \dots, Q_n], P' \Rightarrow P}{\{P'\} T \{result(P', Q_1), result(P', Q_2), \dots, result(P', Q_n)\}'}$$

where P' is an arbitrary wff and the *result* function is defined in Section 4.

Theorem 5.6 (Soundness of frame rule I). *Frame rule I is sound.*

Proof. It is obvious. \square

Example 5.7. For example, from $\{a \wedge c\} T [\neg a \wedge b, a \wedge \neg b]$ and $(a \wedge b \wedge c) \Rightarrow (a \wedge c)$ one can infer that $\{a \wedge b \wedge c\} T \{\neg a \wedge b \wedge c, a \wedge \neg b \wedge c\}$.

$$\frac{\{P\} T [Q_1, Q_2, \dots, Q_n]}{\{P \wedge \bigwedge_{i=1}^n residue(G_i, Q_i)\} T \{G_1, G_2, \dots, G_n\}}$$

where G_1, G_2, \dots, G_n are arbitrary wffs and the *residue* function is defined as follows:

$$residue(G_i, Q_i) = \begin{cases} false & \text{if } G_i \wedge Q_i = false \\ \bigwedge (G_i - Q_i) & \text{otherwise,} \end{cases}$$

where $G_i - Q_i$ denotes the set difference of the sets of literals named in G_i and Q_i . One can prove that $\text{residue}(G_i, Q_i) \wedge Q_i \Rightarrow G_i$. One way to understand this rule is to assume that G_i is to be the postcondition of workflow W when it produces the outcome Q_i . This requires $P \wedge \text{residue}(G_i, Q_i)$ to be the precondition of W . Since any Q_i might be chosen by W during execution, this precondition needs to be strengthened to $P \wedge \bigwedge_{i=1}^n \text{residue}(G_i, Q_i)$.

Theorem 5.8 (Soundness of frame rule II). *Frame rule II is sound.*

Proof. Let $S = P \wedge \bigwedge_{i=1}^n \text{residue}(G_i, Q_i)$. We only need to prove $\text{result}(S, Q_i) \Rightarrow G_i$ for an arbitrary G_i ($i = 1, 2, \dots, n$). We prove this by contradiction.

Suppose $\text{result}(S, Q_i) \Rightarrow G_i$ is false. Then there must exist a truth assignment A such that $A(\text{result}(S, Q_i)) = \text{true}$ but $A(G_i) = \text{false}$. Thus, there must exist a literal l in G_i such that $A(l) = \text{false}$. Consider the following cases:

1. l appears in Q_i , then $\text{result}(S, Q_i) \Rightarrow Q_i \Rightarrow l$, since $A(\text{result}(S, Q_i)) = \text{true}$ we have $A(l) = \text{true}$. This is a contradiction.
2. \bar{l} appears in Q_i , then we have $\text{residue}(G_i, Q_i) \equiv \text{false}$ and thus $S \equiv \text{false}$, the conclusion of frame rule II is a theorem since the workflow's precondition is *false*.
3. Neither l nor \bar{l} appears in Q_i , then $\text{residue}(G_i, Q_i)$, S and thus $\text{result}(S, Q_i)$ all contain literal l . Since $A(\text{result}(S, Q_i)) = \text{true}$, we have $A(l) = \text{true}$. This is another contradiction.

This proves frame rule II is sound. \square

Example 5.9. Suppose a task is specified by $\{a \wedge c\} T [-a \wedge b, a \wedge \neg b]$ where $Q_1 = \neg a \wedge b$, and $Q_2 = a \wedge \neg b$. For $G_1 = \neg a \wedge c$, and $G_2 = a \wedge d$, we have $\text{residue}(G_1, Q_1) = c$ and $\text{residue}(G_2, Q_2) = d$. Using frame rule II, we have the following theorem: $\{a \wedge c \wedge d\} T \{-a \wedge c, a \wedge d\}$.

Example 5.10. For task specification $\{a\} T [-a \wedge b \wedge d]$ and wff $G = \neg a \wedge b \wedge \neg c$, we have $\text{residue}(G, Q) = \neg c$. Thus $\{a \wedge \neg c\} T \{-a \wedge b \wedge \neg c\}$ is a theorem.

Example 5.11. For task specification $\{a\} T [-a \wedge b]$ and wff $G = a$, we have $\text{residue}(G, Q) = \text{false}$. Thus $\{\text{false}\} T \{a\}$ is a theorem. This indicates that there is no starting state in which T can be executed to produce a .

A special form of this rule occurs when $G_i \equiv Q_i$ ($i = 1, 2, \dots, n$). Then $\text{residue}(G_i, Q_i) \equiv \text{true}$ ($i = 1, 2, \dots, n$), and from frame rule II, we can derive the following inference rule:

$$\frac{\{P\} T [Q_1, Q_2, \dots, Q_n]}{\{P\} T \{Q_1, Q_2, \dots, Q_n\}}$$

In order to derive all workflow descriptions that are logically implied by a given workflow description, we introduce the following inference rule, called the *consequence rule*. It states that, given a workflow description, one can strengthen its precondition, and weaken each outcome in its postcondition and get another workflow description. This allows us to adapt the interfaces (pre- and postconditions) of workflows in order to be able to compose them.

$$\frac{\{P\} W \{Q_1, Q_2, \dots, Q_n\}, P' \Rightarrow P, \forall_i (Q_i \Rightarrow Q'_i)}{\{P'\} W \{Q'_1, Q'_2, \dots, Q'_n\}}$$

Theorem 5.12 (Soundness of consequence rule). *Consequence rule is sound.*

Proof. It is obvious. \square

The following theorem indicates that, by using Frame rule I and the consequence rule, one can derive all workflow descriptions that are logically implied by a given task specification. Therefore, Frame rule II is redundant. However, it is convenient to have Frame rule II when one would like to derive a workflow description with given postconditions, while Frame rule I is useful if one would like to derive a workflow description with a given precondition.

Theorem 5.13 (Completeness). *Given a task specification $\{P\} T [Q_1, Q_2, \dots, Q_n]$, Frame rule I and the consequence rule are complete to derive all workflow descriptions that are logically implied by the task specification.*

Proof. See [46]. □

6. Workflow constructs

Tasks are building blocks of workflows. First of all, a workflow can consist of a single task. Since a task is specified with frame semantics, we can infer its corresponding assertional semantics using the frame rules described in Section 5.

We also define the *SKIP* workflow as the identity, the empty workflow. The assertional semantics of a *SKIP* workflow is defined by the following so called *SKIP rule*:

$$\overline{\{P\} \text{SKIP} \{P\}},$$

where P is an arbitrary wff. The *SKIP* rule formalizes the execution semantics of the identity: the execution of a skip workflow does not change the state of the workflow.

Theorem 6.1 (*Soundness of SKIP rule*). *SKIP rule is sound.*

Proof. It is obvious. □

Similar to the skip statement in programming language, it might be convenient to have this identity workflow. For example, in Section 8, our automatic workflow generation algorithm uses a *SKIP* workflow as the initial value for the partial workflow that is generated.

The following constructs allow us to construct workflows from the *SKIP* workflow, single task workflows and other existing workflows that are constructed recursively using these constructs.

6.1. Composition

The composition construct allows one to run two workflows one after another. Given two workflows, W_1 and W_2 , a workflow can be composed by connecting one of W_1 's output edges to W_2 's input edge. The resulted workflow is denoted by; (W_1, i, W_2) (abbreviated by $W_1; W_2$ when W_1 has only one output edge) where i represents the output edge of W_1 that is connected to the input edge of W_2 . The composition construct is illustrated in Fig. 4. The following inference rule, called the *composition rule*, formalizes the execution semantics of the workflow composition construct.

$$\frac{\{P_1\} W_1 \{Q_1, \dots, Q_n\}, \{P_2\} W_2 \{R_1, \dots, R_m\}, Q_i \Rightarrow P_2}{\{P_1\}; (W_1, i, W_2) \{Q_1, \dots, Q_{i-1}, R_1, \dots, R_m, Q_{i+1}, \dots, Q_n\}}$$

Theorem 6.2 (*Soundness of composition rule*). *Composition rule is sound.*

Proof. It is obvious. □

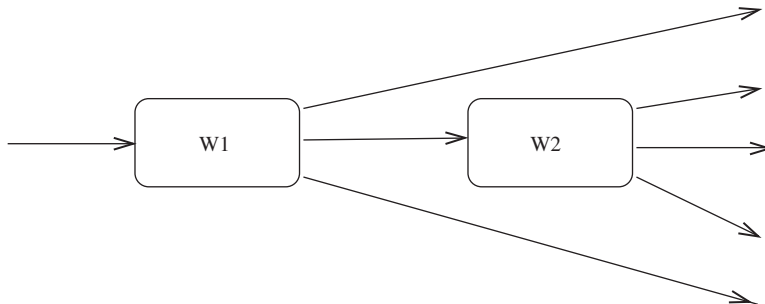


Fig. 4. Composition construct.

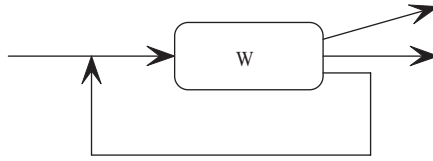


Fig. 5. The loop construct.

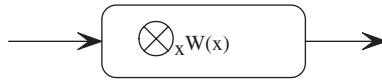


Fig. 6. The universal construct.

Example 6.3. Given two workflow descriptions: $\{\neg a\} W_1 \{b \wedge c, d\}$ and $\{b\} W_2 \{e\}$, the description for one of the compositions is $\{\neg a\}; (W_1, 1, W_2) \{e, d\}$.

6.2. Loop

Given a workflow W described by $P W \{Q_1, \dots, Q_n\}$, if $Q_i \Rightarrow P$, then we can connect the output edge labeled with Q_i to the input edge of W . The resulting workflow is denoted by $\uparrow (W, i)$ and described by $\{P\} \uparrow (W, i) \{Q_1, Q_2, \dots, Q_{i-1}, Q_{i+1}, \dots, Q_n\}$. $\uparrow (W, i)$ will have one input edge and $n - 1$ output edges. The loop construct is illustrated in Fig. 5. The following inference rule, called the *Loop rule*, formalizes the assertional semantics of the loop construct.

$$\frac{\{P\} W \{Q_1, Q_2, \dots, Q_n\}, n \geq 2, Q_i \Rightarrow P}{\{P\} \uparrow (W, i) \{Q_1, \dots, Q_{i-1}, Q_{i+1}, \dots, Q_n\}}$$

Theorem 6.4 (*Soundness of the loop rule*). *The loop rule is sound.*

Proof. see [46]. \square

6.3. Universal

The universal construct allows one to run different instances of the same workflow concurrently.

Given a workflow $W(x)$, a workflow can be constructed that runs all instances of $W(x)$ concurrently for $x \in D \subseteq \text{Dom}(x)$. The workflow is denoted by $\otimes_{x \in D} W(x)$ (abbreviated by $\otimes_x W(x)$ when D is implicit). When $\otimes_x W(x)$ is initiated, all instances of $W(x)$ will be initiated concurrently, and the execution of $\otimes_x W(x)$ finishes when the execution of all instances finish. Workflow $\otimes_x W(x)$ is illustrated in Fig. 6 and it has one input edge and one output edge. The execution semantics of the universal construct is formalized by the following rule, called the *universal rule*.

$$\frac{\{P(x)\} W(x) \{Q_1(x), Q_2(x), \dots, Q_n(x)\}, *}{\{\forall_x P(x)\} \otimes_x W(x) \{\forall_x Q(x)\}},$$

where $P(x), Q_1(x), \dots, Q_n(x)$ are arbitrary wffs and $Q(x) = Q_1(x) \vee \dots \vee Q_n(x)$. Since $W(x)$ itself might be constructed from other tasks, a wff, called *intermediate assertion*, is associated with each intermediate edge of $W(x)$ which is expected to be true for the correctness proof of $W(x)$. The $*$ of the universal rule represents the following interference-free condition: tasks of different instances of $W(x)$ do not interfere with each other's pre- and post-conditions and their intermediate assertions [54].

Theorem 6.5 (*Soundness of universal rule*). *Universal rule is sound.*

Proof. It is obvious. \square

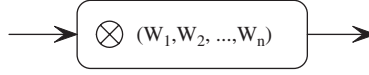


Fig. 7. The conjunction construct.

Example 6.6. Suppose $\{da\} \text{Settle}(x) [stl(x)]$ and $Dom(x) = \{ 'A', 'B' \}$, we can infer $\{da\} \text{Settle}(x) \{stl(x)\}$ using the frame rule and then $\{da\} \otimes_x \text{Settle}(x) \{ \forall_x stl(x) \}$ since the interference-free condition is satisfied.

6.4. Conjunction

The universal construct allows one to run different instances of the same workflow concurrently. A conjunction construct, on the other hand, allows one to run several different workflows concurrently.

Given a set of workflows W_1, W_2, \dots, W_n , a workflow denoted by $\otimes(W_1, W_2, \dots, W_n)$ can be constructed such that during execution, all workflows W_1, W_2, \dots, W_n will execute concurrently. The conjunction construct is illustrated in Fig. 7. The following inference rule, called the *conjunction rule*, formalizes the execution semantics of the workflow conjunction construct.

$$\frac{\{P_i\} W_i \{Q_i^1, \dots, Q_i^{m_i}\} (i = 1, \dots, n), *}{\{\bigwedge_{i=1}^n P_i\} \otimes (W_1, \dots, W_n) \{\bigwedge_{i=1}^n Q_i\}},$$

where $Q_i = Q_i^1 \vee \dots \vee Q_i^{m_i}$ ($i = 1, \dots, n$) and $*$ represents the following interference-free condition: tasks of different W_i do not interfere with each other's pre- and post-conditions and their intermediate assertions [54].

Theorem 6.7 (Soundness of conjunction rule). *Conjunction rule is sound.*

Proof. It is obvious. \square

6.5. Conditional

The conditional construct allows one to execute one instance of a workflow based on the current state.

Given a workflow $W(x, y)$ and a wff $r(x, y)$, if for any value of y , there exists an $x \in Dom(x)$ such that $r(x, y)$ is true, then a workflow can be constructed such that, based on the value of y , will arbitrarily choose a value of $x \in Dom(x)$ such that $r(x, y)$ is true and execute $W(x, y)$. The resulting workflow is denoted by $\oplus_x(r(x, y), W(x, y))$ and illustrated in Fig. 8 (in general, each $W(x, y)$ might have several output edges, and all of them join together). The semantics of the conditional construct can be formalized by the following rule, called *conditional rule*.

$$\frac{\{P(x, y)\} W(x, y) \{Q_1(x, y), \dots, Q_n(x, y)\}, (P \wedge r(x, y)) \Rightarrow P(x, y), gen(y) \Rightarrow \exists_x r(x, y)}{\{P \wedge gen(y)\} \oplus_x (r(x, y), W(x, y)) \{\exists_x Q(x, y)\}},$$

where $Q(x, y) = Q_1(x, y) \vee \dots \vee Q_n(x, y)$.

Theorem 6.8 (Soundness of conditional rule). *The Conditional rule is sound.*

Proof. It is obvious. \square

Example 6.9. Suppose we have two tasks

- Task *Die* allows one to throw a die and save the result in variable y . It is described by $\{true\} \text{Die} \{gen(y)\}$ where $Dom(y) = \{1, 2, 3, 4, 5, 6\}$.
- Task *Init*(x) allows party x to propose the initial bid during a negotiation. It is described by $\{true\} \text{Init}(x) \{newbid(x)\}$, where $Dom(x) = \{1, 2\}$ and $Dom(x)$ contains all the identifiers of the parties involved.

Suppose we want to construct a workflow such that two parties throw a die, if the result is greater than 3, then party 1 will propose the initial bid, otherwise party 2 will propose the initial bid. The workflow is described

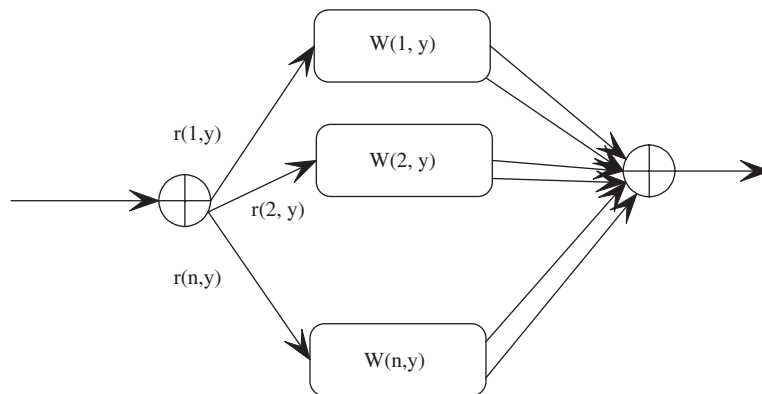


Fig. 8. The conditional construct.

by $\{true\} Die; \oplus_x(r(x, y), Init(x)) \{newbid(x)\}$ where

- $r(1, y) \stackrel{def}{=} y > 3$,
- $r(2, y) \stackrel{def}{=} y \leq 3$.

It is interesting to consider the special case in which $gen(y) \equiv true$, $r(x, y) \equiv true$, $P \equiv \forall_x P(x)$, and $W(x, y)$ is free of y , then the conditional rule becomes:

$$\frac{\{P(x)\} W(x) \{Q_1(x), \dots, Q_n(x)\}}{\{\forall_x P(x)\} \oplus_x W(x) \{\exists_x Q(x)\}},$$

where $Q(x) = Q_1(x) \cdots Q_n(x)$ and $\oplus_x W(x)$ is an abbreviation of $\oplus_x(true, W(x))$.

Example 6.10. For task: $\{da\} Settle(x) \{stl(x)\}$ and $D = \{A', B'\}$, we can infer $\{da\} \oplus_x Settle(x) \{\exists_x stl(x)\}$.

6.6. Disjunction

The conditional construct allows one to execute one instance of the same workflow based on the current state; the disjunction construct, on the other hand, allows one to execute one workflow of a given set of workflows based on the current state.

Given a set of workflows $W_1(y), W_2(y), \dots, W_n(y)$ and a wff $r(x, y)$, if for any value of y , there exists an $x \in Dom(x)$ such that $r(x, y)$ is true, then a workflow can be constructed that, based on the value of y , will arbitrarily choose a value of $x \in Dom(x)$, such that $r(x, y)$ is true and execute $W_x(y)$. The resulting workflow is denoted by $\oplus_x(r(x, y), W_1(y), \dots, W_n(y))$. The disjunction construct is illustrated in Fig. 9. The following rule, called *Disjunction rule* formalizes the semantics of the disjunction construct under this situation.

$$\frac{\{P_x(y)\} W_x(y) \{Q_x^1(y), \dots, Q_x^{m_x}(y)\} (x = 1 \cdots n), (P \wedge r(x, y)) \Rightarrow P_x(y), gen(y) \Rightarrow \exists_x r(x, y)}{\{P \wedge gen(y)\} \oplus_x (r(x, y), W_1(y), \dots, W_n(y)) \{\bigvee_{x=1}^n Q_x(y)\}},$$

where $Q_x(y) = Q_x^1(y) \vee \dots \vee Q_x^{m_x}(y)$ ($x = 1 \cdots n$).

Theorem 6.11 (*Soundness of disjunction rule*). *The Disjunction rule is sound.*

Proof. It is obvious. \square

Similarly, a special case is one in which $gen(y) \equiv true$, $r(x, y) \equiv true$, $P = \bigwedge_{x=1}^n P_x$, and $W_x(y)$ is free of y , then Disjunction rule becomes

$$\frac{\{P_x\} W_x \{Q_x^1, \dots, Q_x^{m_x}\} (x = 1 \cdots n)}{\{\bigwedge_{x=1}^n P_x\} \oplus_x (W_1, \dots, W_n) \{\bigvee_{x=1}^n Q_x\}},$$

where $Q_x = Q_x^1 \vee \dots \vee Q_x^{m_x}$ ($x = 1 \cdots n$).

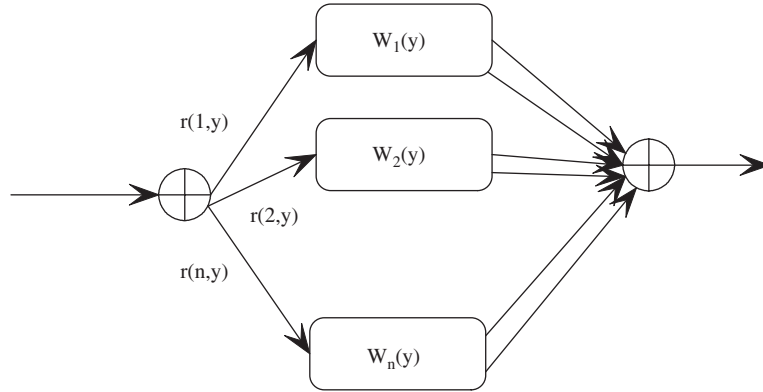


Fig. 9. The disjunction construct.

6.7. Putting it together

In this chapter, in addition to the SKIP workflow and single task workflows, we have defined six workflow constructs that allow us to build workflows from existing workflows. Except the composition construct, all constructs produce a workflow with one-input–one-output property. The following open problems remain to be solved:

- Develop a task and workflow model in which both tasks and workflows have one-input–one-output property, and adapt the composition rule to the new model so that all these six constructs preserve this one-input–one-output property.
- Investigate the completeness of these constructs (i.e., is this set of constructs complete to represent any reasonable workflow). In particular, the notion of *completeness* needs to be formalized, and the completeness proof needs to be investigated.

In this paper, we limit ourselves to consider a workflow space defined recursively by the six constructs we have defined. More formally,

Definition 6.12 (*Definable workflow*). In addition to the SKIP workflow and single task workflows, a workflow is definable by the following rules:

1. If W_1 and W_2 are workflows, then $;$ (W_1, i, W_2) is a workflow.
2. If W is a workflow then \uparrow (W, i) is a workflow.
3. If $W(x)$ is a workflow, then $\otimes_x W(x)$ is a workflow.
4. If W_1, \dots, W_n are workflows, then $\otimes(W_1, \dots, W_n)$ is a workflow.
5. If $W(x, y)$ is a workflow and $r(x, y)$ is a wff, then $\oplus_x(r(x, y), W(x, y))$ is a workflow.
6. If $W_1(y), \dots, W_n(y)$ are workflows and $r(x, y)$ is a wff, then $\oplus_x(r(x, y), W_1(y), \dots, W_n(y))$ is a workflow.
7. Besides the SKIP workflow and single task workflows, workflows can only be constructed from a finite number of applications of the above rules.

Given a workflow, it might or might not be correct with the correctness notion defined in Definition 5.3. The next section sketches an algorithm to check if a workflow is correct with respect to a given workflow precondition and postcondition.

7. Automatic workflow verification

Given a workflow W definable by Definition 6.12, we would like to check automatically if W is correct with respect to a given workflow precondition $WPre$ and workflow postcondition $WPost_1, \dots, WPost_n$, i.e., to check if $\{WPre\} W \{WPost_1, WPost_2, \dots, WPost_n\}$ is a theorem. In this section we sketch such an algorithm.

The algorithm is based on function *Annotate* that marks each edge e of W with a wff, A_e , called its *annotation*, to characterize the state of the workflow when control point reaches e . The pseudocode of *Annotate*($WPre, W$) is

```

Annotate(WPre, W)
{
  Annotate the input edge of W with WPre

  if(W == SKIP) // SKIP workflow
    annotate the output edge of W with WPre
    and return true
  else if(W is a single task workflow){
    suppose W is implemented by {P} T [Q1, Q2, ..., Qn]
    if(WPre ⇒ P)
      annotate each output edge i of T
      by result(WPre, Qi) and return true
  }
  else if(W == ; (W1, i, W2)) // Composition
    if(Annotate(WPre, W1)) return Annotate(Ai(W1), W2)
  }
  else if(W == ↑ (W1, i)) // Loop
    if(Annoate(WPre, W1) and Ai(W1) ⇒ WPre)
      return true
  else if(W == ⊗xW(x)) // Universal
    if(Annotate(WPre, W(x)) and tasks of instances of
      W(x) are interference – free of each other){
      annotate the output edge of W with
      ∀xAD(W(x)) and return true
    }
  }
  else if(W == ⊗(W1, W2, ..., Wn)) // Conjunction
    if(all of Annotate(WPre, Wi) (i = 1, 2, ..., n)
      return true and tasks of all Wi are
      interference – free of each other){
      annotate the output edge of W by ∧i=1n AD(Wi)
      and return true
    }
  }
  else if(W == ⊕x(r(x, y), W(x, y))) // Conditional
    if(Annotate(WPre ∧ r(x, y), W(x, y))){
      annotate the output edge of W
      by ∃xAD(W(x, y)) and return true
    }
  }
  else if(W == ⊕x(r(x, y), W1(y), ..., Wn(y)) // Disjunction
    if(all Annotate(WPre ∧ r(x, y), Wx(y)) return true){
      annotate the output edge of W
      by ∨x=1n AD(Wx(y)) and return true
    }
  }
  }
  return false
}

```

Fig. 10. Function Annotate pseudocode.

illustrated in Fig. 10. We use $A_i(W)$ to denote the annotation of the output edge i of workflow W , and use $AD(W)$ to denote the disjunction of the annotations of all the output edges of workflow W .

Function *Annotate* will abort the annotation process and return false whenever one of the following situations occurs:

1. The annotation of some edge of W does not imply the precondition of the following task.
2. W has a component $\otimes_x W(x)$, and some tasks of instances of $W(x)$ interfere one with another.
3. W has a component $\otimes(W_1, \dots, W_n)$, and some tasks of workflow W_i ($i = 1, \dots, n$) interfere one with another.

Otherwise it will complete the annotation process and return true. A workflow W is correct if and only if

1. $Annotate(WPre, W)$ returns true.
2. The annotation of each output edge of W implies the workflow postcondition $WPost$.

If only 1 holds, then those output edges whose annotations do not imply the workflow postcondition are referred to as W 's *dangling edges*. We describe function *Annotate* in the following.

Given the workflow precondition $WPre$, $Annotate(WPre, W)$ annotates the input edge of W by $WPre$, and annotates other edges of W recursively based on the outmost construct (see Definition 6.12) of W (the pseudocode of $Annotate(WPre, W)$ is shown in Fig. 10 in which “//” is the delimiter for comments):

1. *SKIP workflow*: Annotate the output edge of W with $WPre$ and return true.
2. *Single task workflow*: Suppose W is implemented by $\{P\} T [Q_1, Q_2, \dots, Q_n]$. If $WPre \Rightarrow P$ holds then annotate each output edge i of T with $result(WPre, Q_i)$ and return true; otherwise, return false.
3. *Composition*: $W = ;(W_1, i, W_2)$. Invoke $Annotate(WPre, W_1)$ and if it returns true then invoke $Annotate(A_i(W_1), W_2)$. If both $Annotate(WPre, W_1)$ and $Annotate(A_i(W_1), W_2)$ return true, then return true; otherwise return false.
4. *Loop*: $W = \uparrow(W_1, i)$. Invoke $Annotate(WPre, W_1)$ and if it returns true and $A_i(W_1) \Rightarrow WPre$ then return true; otherwise, return false.
5. *Universal*: $W = \otimes_x W(x)$. Invoke $Annotate(WPre, W(x))$, if it returns true and tasks of all instances of $W(x)$ are interference-free [54], then annotate the output edge of W with $\forall_x AD(W(x))$ and return true; otherwise return false.
6. *Conjunction*: $W = \otimes(W_1, W_2, \dots, W_n)$. Invoke $Annotate(WPre, W_i)$ ($i = 1, 2, \dots, n$) and if all of them return true and tasks of all W_i ($i = 1, \dots, n$) are interference-free [54], then annotate the output edge of W with $\bigwedge_{i=1}^n AD(W_i)$ and return true; otherwise return false.
7. *Conditional*: $W = \oplus_x(r(x, y), W(x, y))$. Invoke $Annotate(WPre \wedge r(x, y), W(x, y))$, and if it returns true then annotate the output edge of W with $\exists_x AD(W(x, y))$ and return true; otherwise, return false.
8. *Disjunction*: $W = \oplus_x(r(x, y), W_1(y), \dots, W_n(y))$. Invoke $Annotate(WPre \wedge r(x, y), W_x(y))$ for each $x = 1, 2, \dots, n$, and if all of them return true then annotate the output edge of W with $\bigvee_{x=1}^n AD(W_x(y))$ and return true; otherwise return false.

An example of workflow automatic verification is available in [46].

8. Automatic workflow generation

In this section, we describe an algorithm that, given a task library L , a workflow precondition $WPre$ and postcondition $WPost$, will automatically generate a correct workflow with respect to $WPre$ and $WPost$. To guarantee the termination of our algorithm, we limit ourselves to generate only workflows that contain no more than $MaxSize$ tasks, where $MaxSize$ is an arbitrary integer specified by the user. We assume that negation in a wff only occurs in literals since we can always transform an arbitrary wff into an equivalent one with this property using De Morgan's laws. We also assume that each wff is fully parenthesized, and its outmost logical connective is called its *principal operator*.

We consider the workflow generation as a search problem in a *search space graph* (See Fig. 11) which is defined as follows: nodes represent sets of states and are labeled by the corresponding wff's, and edges represent workflows and are labeled by the corresponding workflows. Each edge labeled by workflow W connects from node Q_W , the postcondition of W , to node P_W , the precondition of W . Therefore, for each node G , its outgoing edges represent those workflows that have G as the postcondition, and connect to nodes, called G 's *children*, that represent their corresponding preconditions.

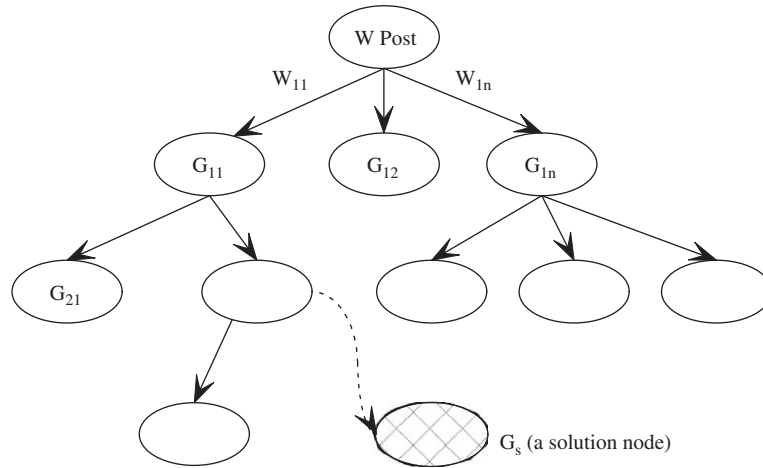


Fig. 11. Search space graph.

For example, in Fig. 11, the outgoing edges of node $WPost$ are labeled by $W_{11}, W_{12}, \dots, W_{1n}$, and the children of $WPost$ are labeled by $G_{11}, G_{12}, \dots, G_{1n}$. Given a task library L , a workflow precondition $WPre$ and postcondition $WPost$, the search space graph must be finite since we assume all variables have finite domains in our workflow model. Loops are possible in a search space graph since we might have the following situation: there exist two nodes G_A and G_B such that there is a path from G_A to G_B , and one of the outgoing edges of G_B corresponds to a workflow whose precondition is G_A . For example, for task library L that contains the following two task specifications, $\{a\} T_1 [b]$ and $\{b\} T_2 [a]$, and workflow postcondition a , the corresponding space graph has a cycle.

To generate a workflow with respect to workflow precondition $WPre$ and postcondition $WPost$, one searches a path in the space graph from the node labeled by $WPost$, to some node G_s (the shaded node in Fig. 11) such that $WPre \Rightarrow G_s$. To avoid getting into an infinite loop, one might keep track of all the nodes that have been visited, and backtrack if a visited node is visited again. Another alternative, which is used in this paper, is to limit ourselves to generate workflows that contain at most $MaxSize$ tasks, where $MaxSize$ is specified by the user. In theory, this might lead to the incompleteness of our workflow generation algorithm, since there might exist a correct workflow that contains more than $MaxSize$ tasks; in practice, however, one can usually estimate a reasonable value for $MaxSize$ based on the task library and the requirement of the application.

The automatic workflow generation algorithm is based on function $genWF$, $genQ$ and fix which are described in the following subsections. To generate a workflow from workflow precondition $WPre$ to workflow postcondition $WPost$, one invokes $genWF(WPre, WPost, WPost, SKIP)$, which will in turn invoke itself recursively and the other two functions, $genQ$ and fix .

8.1. Function $genWF$

Function $genWF(WPre, WPost, G, W)$ (see Fig. 12 for its pseudocode) returns a workflow with respect to a given precondition $WPre$ and postcondition $WPost$, or *false* if it cannot find such a workflow. $genWF$ has four arguments:

- $WPre$, the workflow precondition.
- $WPost$, the workflow postcondition.
- G , the current node to be expanded in the search space graph. Initially, it is equal to $WPost$.
- W , the workflow that corresponds to the path from node G to $WPost$. It is the workflow that has been generated so far and initially, it is the $SKIP$ workflow.

The algorithm starts by expanding $G = WPost$ with invocation $genWF(WPre, WPost, WPost, SKIP)$. Given a node G to be expanded, the following three steps are performed:

- If the partial workflow W is already greater than $MaxSize$, then no further attempt to expand G will be made and *false* is returned (line 3).

```

1) genWF(WPre, WPost, G, W)
2) {
3)   if(size(W) > MaxSize) return false
4)   else if(WPre ⇒ G){
5)     if(fix(WPre, WPost, W)) return W
6)     else return false
7)   }
8)   else{
9)     qG = genQ(G)
10)    while(!isEmpty(qG)){
11)      WG = dequeue(qG)
12)      G' = the precondition of WG w.r.t. G
13)      Ws = genWF(WPre, WPost, G', W')
14)      if(Ws <> false) return Ws
15)    }
16)    return false
17)  }
18) }

```

Fig. 12. Function genWF pseudocode.

- Otherwise, if $WPre \Rightarrow G$, then G is a solution node (line 4). W , the workflow that corresponds to the path from node G to $WPost$, however, might be *incomplete* in the sense that it might contain multiple output edges, and the annotations of some of these output edges might not imply $WPost$. We call these edges as *dangling edges*. $fix(W)$ will transform W into a complete workflow. If $fix(W)$ returns true, then W is completed successfully and is returned, otherwise, W cannot be completed and false is returned (line 13).
- Otherwise, we continue to expand G by invoking $q_G = genQ(G)$ where $genQ(G)$ returns a queue of workflows that have G as its postcondition, and the result is saved in queue q_G (line 9). For each workflow W_G in q_G , we calculate its precondition G' , and continue to expand node G' by invoking $W_s = genWF(WPre, WPost, G', W')$ recursively where $W' = W_G$; W (the while loop from line 10 through 15) until a complete workflow W_s is returned (line 14) or all children of G are expanded (q_G becomes empty in line 10).

8.2. Function genQ

Function $genQ(G)$ returns a queue of workflows that have G as its postcondition and its pseudocode is illustrated in Fig. 13 in which $emptyQueue()$ returns an empty queue, and $q.append(S)$ appends a set of workflows S to the tail of queue q . Queue q is initialized to an empty queue, and we do the following:

1. Add each task T to q if one of its actions contains G , i.e., T alone will implement G (line 4 and 5).
2. *Conjunction* ($G = A \wedge B$) (line 6 through 13): we invoke $q_1 = genQ(A)$ and $q_2 = genQ(B)$ recursively. As a result, q_1 contains a set of workflows that realize A , and q_2 contains a set of workflows that realize B . We then add every possible $\otimes\{W_1, W_2\}$ to q where $W_1 \in q_1$ and $W_2 \in q_2$, and tasks of W_1 and tasks of W_2 are interference-free [54]; we also add every possible $W_1; W_2$ when W_2 does not invalidate A , and every possible $W_2; W_1$ when W_1 does not invalidate B .
3. *Disjunction* ($G = A \vee B$) (line 14 through 18): we invoke $q_1 = genQ(A)$ and $q_2 = genQ(B)$ recursively. As a result, q_1 contains a set of workflows that realize A , and q_2 contains a set of workflows that realize B . We then add every possible $\oplus\{W_1, W_2\}$ to a set S where $W_1 \in q_1$ and $W_2 \in q_2$.
4. *Universal* ($G = \forall_x Q(x)$) (line 19 through 21): we invoke $genQ(Q(x))$ recursively, and add every $\otimes_x W(x)$ into q where $W(x) \in genQ(Q(x))$ if tasks of all instances of $W(x)$ are interference-free of each other [54].
5. *Conditional* ($G = \exists_x Q(x)$) (line 22 and 23): we invoke $genQ(Q(x))$ recursively, and add every $\oplus_x W(x)$ into q where $W(x) \in genQ(Q(x))$.

Finally, function $genQ(G)$ returns q to its caller.

```

1) genQ(G)
2){
3)  q = emptyQueue()
4)  if (an action of some task T contains G)
5)      add every such T to q
6)  else if (G = A ∧ B){ // Conjunction
7)      q1 = genQ(A)
8)      q2 = genQ(B)
9)      q.append({⊗{W1, W2} | W1 ∈ q1, W2 ∈ q2
10)         and tasks of W1 and W2 are interference free})
11)     q.append({W1; W2 | W1 ∈ q1, W2 ∈ q2, and condition 1})
12)     q.append({W2; W1 | W1 ∈ q1, W2 ∈ q2, and condition 2})
13) }
14) else if (G = A ∨ B) // Disjunction
15)     q1 = genQ(A)
16)     q2 = genQ(B)
17)     q.append({⊕{W1, W2} | W1 ∈ q1, W2 ∈ q2)})
18) }
19) else if (G = ∀x Q(x)) // Universal
20)     q.append({⊗x W(x) | W(x) ∈ genQ(Q(x)),
21)         tasks of W(x) are interferece free})
22) else if (G = ∃x (Q(x))){ // Conditional
23)     q.append({⊕x W(x) | W(x) ∈ genQ(Q(x))})
24) }
25) return q
26) }

```

Fig. 13. Function genQ pseudocode.

8.3. Function *fix*

A workflow W is incomplete when the annotation of some output edge of W does not imply the workflow postcondition. Function $fix(WPre, WPost, W)$ (illustrated in Fig. 14) completes a workflow if it is not, and returns true when W is completed, or false if W cannot be completed.

1. If there exists an edge d in W and d 's annotation A_d is implied by the annotation of d 's edge e , A_e , then a loop construct is introduced by connecting e to d .
2. Otherwise, let A_\vee be the disjunction of all the annotations of W , and invoke $W' = genWF(A_e, A_\vee, WPost, SKIP)$. We use A_\vee as the workflow postcondition and use $WPost$ as the initial working wff since any dangling edge whose annotation implies A_\vee can be considered as non-dangling edges during the execution of the above $genWF$, since during the next iteration of the while loop they can be dealt with by the first step. When $genWF$ returns, if W' is not a false then we attach W' to e and we continue to deal with the next dangling edge; otherwise, W cannot be completed, and false is returned.

Finally, after all dangling edges are completed, fix returns true. An example of workflow automatic generation is available in [46].

9. Related work and discussion

Workflow technologies originated from the work on business reengineering and office automation in the 1970s. Since then a substantial effort has been devoted to this area as corporations automate their business processes. Currently, there are hundreds of commercial systems on the market. In [57,49], some of the most typical products are reviewed.

```

1) fix(WPre, WPost, W)
2) {
3)   Annotate(WPre, W)
4)   for(each dangling edge e in W){
5)     if ( $A_e$  implies the annotation  $A_d$  of some edge d)
6)       connect e to d
7)     else{
8)       let  $A_\vee$  be the disjunction of all the annotations of W
8)        $W' = \text{genWF}(A_e, A_\vee, WPost, SKIP)$ 
9)       if ( $W' \llcorner \text{false}$ ) attach  $W'$  to e
10)      else return false
11)    }
12)   Annotate(WPre, W)
13) }
14) return true
15) }
```

Fig. 14. Function fix pseudocode.

Workflows evolved from transactions as the limitations of atomicity and isolation became evident in distributed and heterogeneous systems. Serializability had long been recognized as a performance bottleneck, and database systems all provide less restrictive notions of isolation. On the research side, the nested transaction model [50] was introduced to generalize the concept of atomicity, and the multilevel model [69] was introduced to enhance performance by taking advantage of the semantics of applications. These models, however, preserved the basic concepts of atomicity and isolation.

A more radical departure from the basic transaction model came with extended transaction models, e.g., the ACTA model [22], Flex [55], ConTract [56]. Both atomicity and isolation are relaxed. A review of these extended transaction models are summarized in [30,48], and their application in workflow systems can be found in [38,39]. To deal with advanced applications in which long-lived transactions are present, some researchers propose to decompose transactions into steps to increase concurrency while ensuring semantic correctness [10,41,4].

The workflow model generalizes these models with more constructs and broader functionality [3]. According to the workflow reference model [37] provided by the Workflow Management Coalition, a workflow describes a business process, and a Workflow Management System (WFMS) defines, manages, and executes a workflow using the workflow logic to control the initiation of tasks.

Several formal methods have been proposed for specifying and modeling workflows. These include event algebra [61], state charts [51], Petri nets [65,1], temporal logic [68], and concurrent transaction logic [26]. However, most of these approaches, although formal, assume the workflow is correct if the constraints on data and control flow are satisfied during execution. Whether the final state of the whole workflow is a desired one or not is neither specified nor proved. Since workflows are designed manually in these frameworks, designers are required to understand complicated business requirements and intricate control and data dependencies among different tasks. This becomes an error-prone and time-consuming procedure when a workflow involves hundreds or thousands of tasks. This has motivated recent effort on automatic workflow generation [58,23]. In this paper, we have developed a formal workflow model in which workflows can be constructed from a library of tasks to promote task reuse. The semantics of tasks and workflows is specified in terms of preconditions and postconditions, and a sound inference rule is established to specify the semantics of each workflow construct. Based on this model, an algorithm has been developed to generate a workflow automatically from a task library and a specification of the desired outcome.

Our workflow model currently supports communication between different tasks via the shared *variables* that are stored in a workflow database. However, more advanced event-driven communication between tasks are desirable to support more general business processes. We will investigate the workflow verification and generation problems in such a richer set of primitives in the future.

Observant readers might notice that the constructs we introduced for our workflow model have a lot of similarities to those in programs. However, a workflow has the following two characteristics that distinguishes it from

a program:

- While the building blocks of a program are *statements*, the building blocks of a workflow are *tasks*, which are software components with well-defined input–output interfaces. Therefore, although the constructs for a regular program need to be complete to spell out all possible executions, the design of constructs for workflows only needs to accommodate most control flows in practice.
- In our workflow model, the state space of a workflow is finite and loops take a limited form, as a result, automatic verification and generation become possible. In contrast, a general program cannot be verified or generated automatically due to the need to choose loop invariants for which there exist no general algorithms.

The problem of automatic workflow generation is closely related to the problem of propositional STRIPS planning in artificial intelligence, which is to find a sequence of actions to achieve a goal from a given initial state. The readers are referred to [2] and [36] for surveys on the planning literature. In general, propositional STRIPS planning is intractable [20], hence, in practice, planning systems are built based on various heuristics or restrictions over the expressiveness of actions and formulas [20,52]. Currently, GRAPHPLAN [16] and SATPLAN [43] are considered among the most efficient planning systems.

In contrast to the propositional STRIPS planning model, the workflow domain has the following characteristics:

- In its full generality, in the planning domain, there are few restrictions on actions (e.g., a robot can pick up and put down an object an arbitrary number of times). Therefore, the search space is large and the problem is intractable. We believe in the workflow domain, each task generally achieves a significant, durable result (e.g., update a bank account). If it runs successfully, it generally needs to be run only once (or a bounded number of times) in a workflow execution. In the literature, the *unique-event property* assumption is made for this purpose [26,62]. As a result, the search for a solution is greatly simplified.
- In the planning domain, there are usually no ordering relationships between propositions. In the workflow domain, as each task gets executed, the workflow state should be updated to reflect the “progress” that has been made. In [46], the notion of “progress” is formalized by a hierarchical structure of a task library, i.e., the acyclicity of its dependency graph. This hierarchical structure will significantly reduce the size of the problem search space and may lead to efficient algorithms.
- A workflow may be more than just a sequence of tasks; instead, it may contain workflow constructs such as conditionals and parallels. The semantics of these constructs need to be considered to generate a correct workflow.

The first two characteristics suggest that there might exist efficient algorithms for automatic workflow generation; the last characteristic implies that existing planning algorithms are not directly applicable to automatic workflow generation.

Recently, the emergence of Web service technologies provides a new way for applications to communicate with each other. Web service protocols, such as SOAP and WSDL, provide a foundation that software applications can use to expose their programmatic interfaces on the Web. Many standards are proposed on how to specify the coordination of Web service communication and execution as workflows. Microsoft’s XLANG [64] is a block-structured language with control flow constructs such as *sequence*, *switch*, *while loop*, *all* for parallel routing, and *pick* for race conditions based on timing or external triggers. In contrast to XLANG, IBM’s WSFL [44] is not limited to block structures and allows for directed graphs, which can be nested but must be acyclic. In the meanwhile, Sun, BEA, SAP and Intalio have introduced another Web service composition language called WSCI (Web Service Choreography Interface) [5]. Intalio also initiated the Business Process Management Initiative (BMPI.org) which developed the BPML (Business Process Markup Language) [17]. OASIS and UN/CEFACT support ebXML (Electronic Business using eXtensible Markup Language) (www.ebxml.org). Another notable language is XPDL (XML Process Definition Language) [53] that was released by the Workflow Management Coalition (WfMC) (www.wfmc.org) to support the exchange of workflow specifications between different workflow products. The recently released Business Execution Language for Web Services (BPEF4WS) [25] combines several standards such as WSFL [44] from IBM and XLANG [64] from Microsoft. It allows a mixture of block structured and graph structured process models thus making the language more expressive but more complex at the same time. Finally, some researchers have used UML [29] as a workflow specification language. Having a standard is a good idea, but having too many of them might be not. A comparison between these different languages is beyond the scope of this paper, the reader is referred to [66] for such details.

It is not our goal to introduce another workflow specification language. Instead, our development of a workflow model focuses on supporting semantics based verification and synthesis of workflows. It is expected that the algorithms we presented here can be adapted to verify and synthesize business processes defined using other workflow specification languages as well. For example, a verification and synthesis algorithm for BPEL4WS abstract processes [25] has been developed recently as an extension of the work presented in this paper [28].

Some work has been done on automatic Web service synthesis based on AI planning. For example, Daniela Berardi et al. [8,9] used situation calculus to model the actions of Web services, and dynamically generate an execution tree. Mark Carman et al. [21] maps the synthesis problem to a planning problem in which state descriptions are ambiguous and operator definitions are incomplete. Dan Wu et al. [70] described how to use an AI planning system SHOP2 to automatically compose Web services based on DAML-S. Mithun Sheshagiri [59] translates a DAML-S description of Web services into a set of STRIPS-like planning operators by a set of rules and queries.

Though most approaches map Web service synthesis into AI planning problems, many point out that the Web service domain has its own special characteristics and current AI planning is not going to provide a complete solution [63,59]. First, a Web service (task) often has multiple outcomes and non-deterministic behavior. The workflow needs to guarantee correctness on all paths. Second, workflow languages provide a set of control constructs, and the synthesis algorithm needs to choose the constructs wisely in order to produce a well-structured workflow. Planning focuses on generating a path for conjunctive goals and does not consider actions with conditional or non-deterministic effects. Recently, planning algorithms have been developed to handle more complex situations, such as conditions and non-determinism (i.e., [35,24]). However, they are not intended to generate structured workflows and many workflow constructs and concepts are missing.

10. Conclusion

We defined a set of inference rules for the workflow constructs that we introduced. These rules are the basis of reasoning about the correctness of workflows and automatic generation of workflows based on the workflow description and a given task library. Future work includes the improvement of these algorithms and their applications, for example, the virtual enterprises and failure handling. We will also investigate the workflow verification and generation problems in a richer set of primitives that involve event-driven communication between tasks. Future work also includes a performance comparison study of the proposed workflow synthesis algorithm and existing planners such as PDDL planner [33].

Acknowledgements

We would like to thank Ziyang Duan for the numerous stimulating discussions that help improve the quality of this paper. We would also like to thank the anonymous reviewers for their constructive suggestions and helpful comments.

References

- [1] N.R. Adam, V. Atluri, W.-K. Huang, Modeling and analysis of workflows using petri nets, *J. Intelligent Information Systems, Special Issue on Workflow* 10 (2) (1998) 131–158.
- [2] J. Allen, J. Hendler, A. Tate, *Readings in Planning*, Morgan Kaufman Publishers, San Mateo, CA, 1990.
- [3] G. Alonso, D. Agrawal, A.E. Abbadi, M.U. Kamath, R. Ginthvr, C. Mohan, Advanced transaction models in workflow contexts, in: *Proc. Twelfth Internat. Conf. Data Engineering*, February 1996, pp. 42–48.
- [4] P. Ammann, S. Jajodia, I. Ray, Applying formal methods to semantic-based decomposition of transactions, *ACM Trans. Database Syst.* 22 (2) (1997) 215–254.
- [5] A. Arkin, Web service choreography interface 1.0, 2002 <<http://www.w3.org/TR/wsci/>>.
- [6] I.B. Arpinar, U. Halici, S. Arpinar, A. Dogac, Formalization of workflows and correctness issues in the presence of concurrency, *Distributed and Parallel Databases* 7 (2) (1999) 199–248.
- [7] P.C. Attie, M.P. Singh, A. Sheth, M. Rusinkiewicz, Specifying and enforcing intertask dependencies, in: *Proc. 19th Internat. Conf. on Very Large Data Bases*, Dublin, Ireland, August 1993, pp. 134–145.
- [8] D. Berardi, D. Calvanese, G. Giacomo, M. Mecella, Reasoning about actions for e-service composition, in: *ICAPS 2003 Workshop on Planning for Web Services*, Trento, Italy, June 2003.

- [9] D. Berardi, D. Calvanese, D.G. Giuseppe, M. Lenzerini, M. Mecella, Automatic composition of e-services that export their behavior, in: *The First Internat. Conf. on Service Oriented Computing*, Trento, Italy, December 2003.
- [10] A. Bernstein, D. Gerstl, P. Lewis, Concurrency control for step-decomposed transactions, *Inform. Syst.* 24 (8) (1999).
- [11] A.J. Bernstein, D.S. Gerstl, W.H. Leung, P.M. Lewis, Design and performance of an assertional concurrency control system, in: *Proc. Internat. Conf. on Data Engineering*, 1998.
- [12] A.J. Bernstein, D.S. Gerstl, P.M. Lewis, A concurrency control for step-decomposed transactions, *Inform. Syst.* 24 (8) (1999).
- [13] A.J. Bernstein, D.S. Gerstl, P.M. Lewis, S. Lu, Using transaction semantics to increase performance, in: *Eighth Internat. Workshop on High Performance Transaction Systems*, Pacific Grove, California, USA, September 1999.
- [14] A.J. Bernstein, P.M. Lewis, High performance transaction systems using transaction semantics, *Distributed and Parallel Databases* 4 (1) (1996).
- [15] A.J. Bernstein, P.M. Lewis, S. Lu, Semantic conditions for correctness at different isolation levels, in: *Proc. 16th IEEE Internat. Conf. on Data Engineering*, San Diego, CA, USA, March 2000, pp. 57–66.
- [16] A. Blum, M. Furst, Fast planning through planning graph analysis, *Artificial Intelligence* 90 (1–2) (1997) 281–300.
- [17] BPMI.org, The business process modeling language (BPML), 2002 <<http://www.bpml.org/bpml.jsp>>.
- [18] Y. Breitbart, A. Deacon, H.-J. Schek, A. Sheth, G. Weikum, Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows, *SIGMOD Record* 22 (3) (September 1993) 23–30.
- [19] Y. Breitbart, H. Garcia-Molina, A. Silberschatz, Overview of multidatabase transaction management, *VLDB J.* 1 (2) (1992) 181–240.
- [20] T. Bylander, The computational complexity of propositional STRIPS planning, *Artificial Intelligence* 69 (1–2) (1994) 165–204.
- [21] M. Carman, L. Serafini, P. Traverso, Web service composition as planning, in: *ICAPS 2003 Workshop on Planning for Web Services*, Trento, Italy, June 2003.
- [22] P. Chrysanthis, K. Ramamritham, ACTA: A framework for specifying and reasoning about transaction structure and behavior, in: *Proc. ACM-SIGMOD Internat. Conf. on Management of Data*, 1990, pp. 194–203.
- [23] S. Chun, V. Atluri, N.R. Adam, Domain knowledge-based automatic workflow generation, in: *Proc. 13th Internat. Conf. on Database and Expert Systems Applications (DEXA)*, Aix en Provence, France, September 2002.
- [24] A. Cimatti, M. Roveri, P. Traverso, Automatic OBDD-based generation of universal plans in non-deterministic domains, in: *Proc. 15th National Conf. on Artificial Intelligence*, Wisconsin, 1998, pp. 875–881.
- [25] F. Curbera, Y. Golland, J. Klein, F. Leymann, D. Roller, S. Thatte, S. Weerawarana, Business process execution language for web services, Version 1.1, 2003 <<http://www-106.ibm.com/developerworks/library/ws-bpel/>>.
- [26] H. Davulcu, M. Kifer, C.R. Ramakrishnan, I.V. Ramakrishnan, Logic based modeling and analysis of workflows, in: *Proc. ACM Symp. on Principles of Database Systems*, Seattle, WA, USA, June 1998, pp. 25–33.
- [27] U. Dayal, M. Hsu, R. Ladin, Organizing long-running activities with triggers and transactions, in: *Proc. ACM SIGMOD Conf. on the Management of Data*, Atlantic City, New Jersey, USA, May 1990, pp. 204–214.
- [28] Z. Duan, A. Bernstein, P. Lewis, S. Lu, Semantics based verification and synthesis of BPEL4WS processes, in: *Proc. IEEE Internat. Conf. on Web Services*, San Diego, CA, USA, July 2004, pp. 734–737.
- [29] M. Dumas, A.H.M. ter Hofstede, Uml activity diagrams as a workflow specification language, in: *Proc. Fourth Internat. Conf. on the Unified Modeling Language*, Lecture Notes in Computer Science, Vol. 2185, Toronto, Canada, October 2001, pp. 76–90.
- [30] A. Elmagarmid (Ed.), *Database Transaction Models For Advanced Applications*, Morgan Kaufmann Publishers, Los Altos, CA, 1992.
- [31] H. GarciaMolina, D. Gawlick, J. Klein, K. Kleissner, K. Salem, Modeling long-running activities as nested sagas, *IEEE DB Eng. Bull.* 14 (1) (1991).
- [32] D. Georgakopoulos, M. Hornick, P. Krychniak, F. Manola, Specification and management of extended transactions in a programmable transaction environment, in: *Proc. 10th Internat. Conf. on Data Engineering*, Houston, TX, February 1994, pp. 462–477.
- [33] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, D. Wilkins, Pddl-the planning domain definition language, Technical Report, Yale University, New Haven, CT, 1998.
- [34] J. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques*, Corrected Second Printing, Morgan Kaufman Publishers, Los Altos, CA, 1993.
- [35] E. Hansen, S. Zilberstein, LAO*: A heuristic search algorithm that finds solutions with loops, *Artificial Intelligence* 129 (1–2) (2001) 35–62.
- [36] J. Hendler, A. Tate, M. Drummond, AI planning: Systems and techniques, *AI Magazine* 11 (2) (1990) 61–77.
- [37] D. Hollingsworth, The workflow reference model, Workflow Management Coalition TC00-1003, January 1995 <<http://www.wfmc.org/standards/docs/tc003v11.pdf>>, last visited on July 14, 2002.
- [38] M. Hsu, Letter from the special issue editor, *Bull. Technical Committee on Data Eng.* 16 (2) (1993) 3.
- [39] M. Hsu, Letter from the special issue editor, *Bull. Technical Committee on Data Eng.* 18 (1) (1995) 2.
- [40] M.R.A. Huth, M.D. Ryan (Eds.), *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge University Press, Cambridge, 2000.
- [41] S. Jajodia, I. Ray, P. Ammann, Implementing semantic-based decomposition of transactions, in: *Proc. Internat. Conf. on the Advanced Information Systems Engineering (CAISE)*, Barcelona, Catalonia, Spain, June 1997.
- [42] M. Kamath, K. Ramamritham, Correctness issues in workflow management, *Distributed Syst. Eng. J.* 3 (4) (1996) 213–221.
- [43] H. Kautz, D. McAllester, B. Selman, Encoding plans in propositional logic, in: *Proc. Fifth Internat. Conf. on the Principle of Knowledge Representation and Reasoning (KR'96)*, Cambridge, MA, November 1996, pp. 374–384.
- [44] F. Leymann, Web services flow language (WSFL 1.0), 2001 <<http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>>.
- [45] F. Leymann, D. Roller, Business process management with flowmark, in: *Proc. 39th IEEE Computer Society Internat. Conf. (CompCon)*, February 1994, pp. 230–234.
- [46] S. Lu, Semantic correctness of transactions and workflows, Ph.D. Thesis, State University of New York at Stony Brook, May 2002.
- [47] S. Mehrotra, R. Rastogi, Y. Breitbart, H.F. Korth, A. Silberschatz, The concurrency control problem in multidatabases: characteristics and solutions, in: *Proc. ACM SIGMOD Conf. on the Management of Data*, San Diego, CA, June 1992, p. 288.

- [48] C. Mohan, Tutorial: Advanced transaction models—survey and critique, ACM SIGMOD Internat. Conf. on Management of Data, Minneapolis, May 1994.
- [49] C. Mohan, Recent trends in workflow management products, standards and research, in: F.N. Afrati, P. Kolaitis (Eds.), Proc. NATO Advanced Study Institute (ASI) on Workflow Management Systems and Interoperability, Istanbul, August 1997, NATO ASI Series, Series F: Computer and Systems Sciences, Vol. 164, Springer, Berlin, 1998.
- [50] J. Moss, Nested Transactions: An Approach to Reliable Computing, MIT Press, Cambridge, MA, 1985.
- [51] P. Muth, D. Wodtke, J. Weissenfels, G. Weikum, A.K. Dittrich, Enterprise-wide workflow management based on state and activity charts, Lecture Notes on Computer Science: Advances in Workflow Management Systems and Interoperability, Springer, Berlin, 1997.
- [52] B. Nebel, On the compilability and expressive power of propositional planning, J. Artificial Intelligence Res. 12 (2000) 271–315.
- [53] R. Norin, Workflow process definition interface—XML process definition language, 2002 <http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf>.
- [54] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs I, Acta Informatica 6 (1976) 319–340.
- [55] M. Reichert, P. Dadam, ADEPT_{flex}—supporting dynamic changes of workflows without losing control, J. Intelligent Inform. Syst., Special Issue on Workflow 10 (2) (1998) 93–129.
- [56] A. Reuter, F. Schwenkreis, Contracts—a low-level mechanism for building general purpose workflow management systems, Bull. Technical Committee on Data Eng. 18 (1) (1995) 4.
- [57] T. Schael, Workflow management systems for process organisations, Lecture Notes in Computer Science, Vol. 1096, 1998.
- [58] H. Schuster, D. Georgakopoulos, A. Cichocki, D. Baker, Modeling and composing service-based and reference process based multi-enterprise processes, in: Proc. Internat. Conf. on the Advanced Information Systems Engineering (CAiSE), Stockholm, Sweden, June 2000.
- [59] M. Sheshagiri, M. desJardins, T. Finin, A planner for composing services described in daml-s, in: ICAPS 2003 Workshop on Planning for Web Services, Trento, Italy, 2003.
- [60] A. Sheth, M. Rusinkiewicz, On transactional workflows, Bull. Technical Committee on Data Eng. 16 (2) (1993) 37.
- [61] M.P. Singh, Semantical considerations on workflows: algebraically specifying and scheduling intertask dependencies, Fifth Internat. Workshop on Database Programming Languages (DBPL), Gubbio, Umbria, Italy, September 1995.
- [62] M.P. Singh, Synthesizing distributed constrained events from transactional workflow specifications, in: Proc. 12th Internat. Conf. on Data Engineering, February 1996, pp. 616–623.
- [63] B. Srivastava, J. Koehler, Web service composition—current solutions and open problems, in: ICAPS 2003 Workshop on Planning for Web Services, Trento, Italy, June 2003.
- [64] S. Thatte, XLANG: Web services for business process design, 2001 <http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm>.
- [65] W. van der Aalst, Verification of workflow nets, in: Proc. Internat. Working Conf. on Information and Process Integration in Enterprises Cambridge, MA, 1996, pp. 179–201.
- [66] W. van der Aalst, Don't go with the flow: web services composition standards exposed, IEEE Intelligent Systems, Jan/Feb 2003, 2003.
- [67] H. Wachter, A. Reuter, The contract model, in: Elmagarmid Ref. ce:cross-ref[30], 1992, pp. 220–263.
- [68] J. Wainer, Logic representation of processes in work activity coordination, ACM Symp. on Applied Computing Villa Olmo, Como, Italy 2000, March 2000.
- [69] G. Weikum, H. Schek, Multi-level transactions and open nested transactions, IEEE Data Eng. Bull., March 1991.
- [70] D. Wu, B. Parsia, E. Sirin, J. Hendler, D. Nau, Automating DAML-S web services composition using SHOP2, in: Proc. Second Internat. Semantic Web Conf. (ISWC2003), Florida, 2003.