15

An Extended Petri Net Model for Supporting Workflows in a Multilevel Secure Environment

Vijayalakshmi Atluri and Wei-Kuang Huang
Center for Information Management, Integration, and Connectivity (CIMIC)
and
MS/CIS Department, Rutgers University, Newark, NJ 07102, U.S.A.
{atluri,waynexh}@andromeda.rutgers.edu

Abstract

This paper makes three contributions to the area of *multilevel secure* (MLS) *workflow management systems* (WFMS). First, it proposes a multilevel secure workflow transaction model. This model identifies the task dependencies in a workflow that cannot be enforced in order to meet multilevel security constraints. Second, it shows how *Petri nets*, a mathematical as well as a graphical tool, can be used to represent various types of task dependencies. Third, it extends the original Petri net (PN) model by proposing a *Secure Petri Net* (SPN) that can automatically detect and prevent all the task dependencies that violate security. This paper then presents algorithms to construct and execute MLS workflow transactions.

1 INTRODUCTION

Workflows typically represent processes involved in manufacturing and office environments and heterogeneous database management systems. In a workflow, the various activities in the process are separated into well defined tasks. These tasks in turn are usually related and dependent on one another, and therefore need to be executed in a coordinated manner.

It has been recognized in (Georgakopoulos et al. 1993, Georgakopoulos, Hornick & Sheth 1995) that, to ensure correctness and reliability, every workflow must be associated with a transaction

model. Transactions traditionally are characterized by simple application logic and short duration that typically execute within few minutes or seconds. Traditional transactions are built on the concepts of atomicity, consistency, isolation and durability (ACID properties). Although the traditional transaction concept can be useful in applications such as airline reservation systems, banking systems and electronic funds transfer, it is inadequate to model the dependencies and other semantic relationships among various tasks existing in workflows. As a result, the traditional transaction model has been extended to capture the task dependencies within a workflow transaction. For example, see the Extended Transaction Model (ETM) proposed in (Georgakopoulos et al. 1993).

The tasks within a workflow are usually related and dependent on one another. These task dependencies are known as intra-workflow dependencies. Thus a workflow transaction TW can be represented as a *partially ordered* set of tasks $tw_1, tw_2, \ldots, tw_n$. In addition, dependencies exist among tasks that belong to different workflows, which are known as inter-workflow dependencies. As advances in workflow management take place, they are also required to support multilevel security.

Security is concerned with the ability to enforce a security policy governing the disclosure, modification or destruction of information. The basic model of multilevel security was first introduced by Bell and LaPadula (1976). The Bell-LaPadula (BLP) model is stated in terms of *objects* (that hold data such as a file or a record) and *subjects* (active entities that manipulate objects). Every object is assigned a classification and every subject a clearance. Classifications and clearances are collectively known as security classes (or levels) and are partially ordered. The BLP model comprises of the following two properties: (1) *simple-security property:* a subject is allowed to read an object only if the former's security level is identical or higher than the latter's security level (no read-up) and (2) *⋆-property:* a subject is allowed to write an object only if the former's security level is identical or lower than the latter's security level (no write-down). (For integrity reasons, most systems do not allow write-ups.) These two restrictions are intended to ensure that there is no flow of information from higher level objects to subjects at lower security levels (Denning 1982). Although they prevent direct flow of information from *high* to *low*⋆, they are not sufficient to ensure that security is not compromised since it could be possible that leakage of information can occur through indirect means via *covert channels.* Covert channels are paths not normally meant for information flow that could nevertheless be used to signal information. They could occur as a subject at a higher security level delaying or aborting another subject at a lower security level.

In the following, we provide an example of a multilevel secure workflow transaction.

Example 1 Consider a workflow transaction that computes the weekly pay of all employees at the end of each week. This process involves several tasks as follows. Task tw_1: compute the number of hours worked by an employee (h) which is the sum of regular hours worked (n) and overtime hours worked (o) by the employee during that week, Task tw_2: calculate the weekly pay of an employee (p) by multiplying h with the hourly rate of the employee (r), and Task tw_3: after computing the pay for the week, reset h, n and o to zero. The information about hourly rate (r) and weekly pay (p) are considered sensitive, and therefore are classified *high*, while the rest of the information is classified *low*. According to the two BLP restrictions, since tw_1 and tw_3 write objects at *low* (h, n and o) they must be *low* tasks, and since tw_2 reads the *high* object (r) and

---

⋆Often, we use the terms *high* and *low* in our discussion to represent two security levels, where *high* is greater than *low* in the partial order.

writes the *high* object (p), it must be a *high* task. Moreover, the following task dependencies exist among tw_1, tw_2 and tw_3: tw_2 can begin only after tw_1 commits, and tw_3 can begin only after tw_2 commits. As seen in this example, a workflow transaction may consist of tasks at different security levels.   □

In this paper, we first propose a *multilevel secure workflow transaction model* in which we identify the allowable and non-allowable task dependencies that lead to potential covert channels. Then, we show how *Petri Nets* (PNs) can be used to model MLS workflow transactions. In addition, we extend the traditional PN to represent the security level of a task, which we refer to as *Secure Petri Net* (SPN). We then demonstrate how SPN can be used to identify and eliminate the task dependencies that do not satisfy the MLS constraints.

1.1   Our Approach

In this section we will first justify why PN is an appropriate modeling tool for representing workflows and then review prior research in using PNs in similar environments. PNs have been extensively studied and used in modeling, specification, validation, performance analysis, control, and simulation of transaction systems.

There are a number of reasons that make PNs an appropriate model to represent workflows. (1) PNs are a graphical as well as a mathematical modeling tool. As a graphical tool, PNs provide visualization (similar to flow charts, block diagrams, and the like) of the workflow process. As a mathematical tool, PNs enable analysis of the behavior of the workflow. For example, the safety of a workflow (i.e., a workflow will terminate in one of the specified acceptable termination states) can be examined by testing for reachability (see section 4 for a definition) of PN. Similar representations include state transition diagrams (Rusinkiewicz & Sheth 1994). Unlike a static state transition diagram, PN is live in the sense that it is capable of capturing the dynamic behavior of any system. It can visualize and represent all properties, relations and restrictions in a workflow such as parallelism, concurrency, synchronization, control flow dependency and temporal relations. (2) PNs are even capable of modeling priorities, concurrent reader-writer, and mutual exclusion, which are relevant in multilevel secure transaction processing where a lower level task must always be prioritized over higher level tasks. (3) Moreover, PN is self-explanatory. If the system is modeled properly, no further verbal description is needed to aid in describing the workflow. (4) Furthermore, the theoretical results are plentiful; the properties of PNs have been and still being extensively studied (Murata 1989). (5) With respect to execution of workflows, ECA (Event-Condition-Action) rules have been used by other researchers. For example, the Extended Transaction Model (ETM) proposed in (Georgakopoulos et al. 1993) uses a combination of rules and conventional transaction management mechanisms such as schedulers, where rules are primarily used to implement the task dependencies and schedulers to enforce correctness and reliability. With respect to implementation, PN can be modeled at a conceptual level and can easily be tied into the design specification and algorithms. The final PN can be treated as a test bed where the system can be simulated and validated before proceeding to detailed design and implementation. (6) Moreover, modification on PNs is relatively simple.

Several researchers have used PNs in transaction processing. For example, in (Elmagarmid, Leu, Litwin & Rusinkiewicz 1990), Elmagarmid et al. describe a scheduler for Flexible Transac-

tions (Elmagarmid 1992)[†] that uses PNs to identify the set of subtransactions schedulable in a given state. They use a special class of PNs called Predicate Petri Nets (PPNs) to capture the precedence relationships among subtransactions and represent each of them as a predicate for each subtransaction.

As in (Elmagarmid et al. 1990), we use PNs to execute MLS workflow transactions and show how several types of control-flow dependencies among the various tasks in it can be modeled using PNs. Our work goes well beyond that of (Elmagarmid et al. 1990) in several aspects. (1) Since (Elmagarmid et al. 1990) models the control-flow dependencies as a single predicate, the PPN does not provide a true visual representation of the dynamic behavior of the workflow (one of the main reasons for using PNs). Our model decomposes each task and represent it as a set of states and transitions. Thus it can explicitly specify the dependencies based on the task primitives. (2) Our model is consistent with the traditional PN model, thereby enabling us to adapt the well established analysis techniques into our work. (3) We extend the Petri net model to incorporate multilevel security, which prevents all task dependencies that cause potential covert channels. This feature is useful for concurrent scheduling of workflows especially when workflows are ad hoc in nature. (4) While modeling single tasks using PNs, we define dependencies and security levels in a more general form so that they can be applied as building blocks to compose large workflow system.

This paper is organized as follows. In sections 2 and 3, we present the workflow transaction model and its multilevel secure counterpart, respectively. In section 4, we give a brief overview of PNs. In section 5, we give the PN model for different types of task dependencies, present our SPN, and propose a mechanism for eliminating covert channels. In section 6 we present algorithms for construction of SPNs and for execution of MLS workflow transactions. Finally, section 7 presents conclusions and some future research we intend to pursue in this area.

2   WORKFLOW TRANSACTION MODEL

In (Rusinkiewicz & Sheth 1994), three types of task dependencies in a workflow have been identified to control the coordination among different tasks. (1) *Control flow dependencies:* These are specified based on the task primitives such as begin, commit and abort of a task. An example of such dependency is "task tw_i can begin only if task tw_j has committed." (2) *Value dependencies:* These are specified such that a task can be controlled based on the output value generated by another task. These dependencies are of the form, "if the output of tw_i is equal to x, then begin tw_j" or "tw_j can begin if tw_i is a success (semantically)."[‡] (3) *External dependencies:* They control the execution of tasks through external variables. Examples include a task tw_i can start its execution only at 9:00am or task tw_j can start execution only 24hrs after the completion of task tw_k.

The task dependencies can either be *static* or *dynamic* in nature. In the static case, the workflow transaction is defined well in advance to its actual execution, whereas dynamic dependencies develop as the workflow progresses through its execution (Sheth, Rusinkiewicz & Karabatis 1993).

[†] A flexible transaction is specified as a set of partially ordered subtransactions.
[‡] Failure of a task does not necessarily mean abort of a task. A task may still semantically fail even if it successfully commits.

Task dependencies may exist among tasks within a workflow transaction (*intra-workflow*) or between two different workflow transactions (*inter-workflow*).

In this paper, we concentrate only on control flow dependencies based on the task primitives. Control-flow dependencies may even pass data to other tasks; we identify them as control-flow dependencies with data-flow.

2.1   Control-flow Dependencies

A control-flow dependency is of the form:

A task tw_j can enter state st_j only after task tw_i enters state st_i.

The state of a task can be expressed in terms of task management primitives such as begin, commit and abort. Thus, execution of a task, in addition to invoking operations on data items, requires invocation of these task management primitives. Control flow dependencies can be modeled based on the ACTA framework (Chrysanthis 1991). Given two tasks tw_i and tw_j in a workflow transaction, a list of possible control-flow dependencies are presented below.

1. Strong Commit Dependency: A task tw_j commits only if tw_i commits (represented as $tw_i \xrightarrow{c} tw_j$).
2. Abort Dependency: A task tw_j must abort if tw_i aborts (represented as $tw_i \xrightarrow{a} tw_j$).
3. Termination Dependency: A task tw_j can terminate (either commit or abort) only after the completion (commit or abort) of tw_i (represented as $tw_i \xrightarrow{t} tw_j$).
4. Begin Dependency: A task tw_j cannot begin until tw_i has begun (represented as $tw_i \xrightarrow{b} tw_j$).
5. Begin-on-Commit Dependency: A task tw_j cannot begin until tw_i commits (represented as $tw_i \xrightarrow{bc} tw_j$).
6. Group Commit: Given any two tasks tw_i and tw_j, either both tw_i and tw_j commit or neither commits (Biliris, Dar, Gehani, Jagadish & Ramamritham 1994).[§] (represented as $tw_i \xrightarrow{gc} tw_j$ or $tw_j \xrightarrow{gc} tw_i$).

A comprehensive list of task dependencies based on these three task primitives, namely, begin, commit and abort, can be found in (Elmagarmid 1992, Chrysanthis 1991), which include commit, weak-abort, exclusion, force-commit-on-abort, serial, begin-on-abort and weak-begin-on-commit dependencies.

2.2   Control-flow Dependencies with Data-flow:

A control-flow dependency with data-flow is of the form:

A task tw_j can enter state st_j only after task tw_i enters state st_i and tw_i passes values of data objects to tw_j.

In these dependencies, in addition to the control flow, there could even be informationflow (or data flow) between the tasks where a task needs to wait for data from another task. Notice that

---

[§]Group commit involving a set of tasks can be defined using pairwise group dependencies.

control-flow dependency with data-flow is meaningful only for limited combinations of st_i and st_j. For example, st_i and st_j can be "commit" and "begin," respectively, but cannot be "begin" and "commit."

3 MULTILEVEL SECURE WORKFLOW TRANSACTION MODEL

In a multilevel secure workflow, a workflow transaction may consist of tasks of different security levels (as in example 1). Thus, the dependency graph consists of nodes at different security levels where the dependency edges may connect tasks of either the same security level or different security levels, which can be distinguished as follows. The dependency edge connecting tasks of the same security level is referred to as *intra-level dependency* and that connecting tasks of different security levels as *inter-level dependency*. Since intra-level dependencies by themselves cannot violate any multilevel security constraints and are no different from the task dependencies in a non-secure environment, hereafter we concentrate only on inter-level dependencies. We further divide inter-level dependencies into two categories: *high-to-low*[¶] and *low-to-high* since their treatment has to be different in a MLS environment because of its "no downward information flow" requirement.

Example 2 Returning to example 1, task tw_2 can begin only after tw_1 commits, thus $tw_1 \xrightarrow{bc} tw_2$, and tw_3 can begin only after tw_2 commits, i.e., $tw_2 \xrightarrow{bc} tw_3$, as shown in figure 1. Both $tw_1 \xrightarrow{bc} tw_2$ and $tw_2 \xrightarrow{bc} tw_3$ are inter-level dependencies where the former is a *low-to-high* and the latter *high-to-low*. ☐
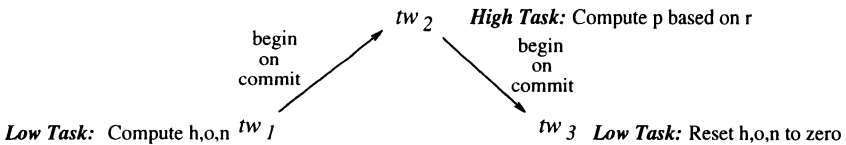

Figure 1 Inter-level dependencies of the multilevel workflow transaction in example 1

Correct execution of a workflow transaction involves (1) enforcing all intra-task and inter-task dependencies, (2) assuring correctness of interleaved execution of multiple workflows, and (3) preserving atomicity of a transaction. Indeed, satisfying each of the above three criteria may conflict with the constraints imposed by multilevel security. In this paper, we focus only on the first criterion, i.e., enforcing the task dependencies.

Enforcing a *low-to-high* dependency will not result in violation of security. However, although one cannot directly enforce *high-to-low* dependencies without compromising security, in some cases, we can simulate their effect. We explain this with example 1 by considering $tw_2 \xrightarrow{bc} tw_3$. The intention of this dependency is to avoid tw_3 to overwrite data that tw_2 has yet to read. Since

---

[¶]Although we use the term *high-to-low*, this dependency also includes those among two incomparable security levels.

we cannot delay tw_3 until tw_2's commit, we may keep an old version of all data that tw_3 updates. Thus we avoid delaying tw_3's begin by providing an old version to the *high* task tw_2. Or, one can redesign the workflow in a clever way as in (Blaustein, Jajodia, McCollum & Notargiacomo 1993) such that no *high-to-low* dependencies exist in the workflow itself. It is important to note that this type of simulation (or even redesign) may not be possible with all types of dependencies. Since enforcing a *high-to-low* dependencies may introduce covert channels, a secure WFMS must identify and prevent such dependencies.

One way of dealing with the covert channels is to reduce the bandwidth of the channel. Another way is to completely eliminate enforcing the *high-to-low* dependencies. In the first case, the following mechanism can be used to enforcing *high-to-low* dependencies.

An approach is to use a buffer at *high* (assume its size is sufficiently large) in which the commit message of the *high* task is stored. This message will first be subjected to a delay of some random duration, and then will be transmitted to *low*. If several such messages of a single workflow transaction get accumulated during the delay period of the first message, these messages cannot be sent at the same time, but must be sent individually with the delay incorporated in between each of them. Thus, though there exists a channel of downward information flow, the bandwidth of this channel would be low. It is important to note that for a system to be secure, (at B3 or A1 level) it is not required to completely eliminate the covert channels but their bandwidth should not exceed 100 bits per second.

In this paper, we take the second approach and provide a protocol to prevent all *high-to-low* dependencies. Our approach uses a PN representation of the task dependencies and detects and prevents all *high-to-low* dependencies. In the next section we give a brief overview of PNs.

4   OVERVIEW OF PETRI NETS

A *Petri Net* (PN) is a bipartite directed graph consisting of two kinds of nodes called *places* and *transitions* where arcs (edges) are either from a place to a transition or from a transition to a place. While drawing a PN, places are represented by circles and transitions by bars. A *marking* may be assigned to places. If a place p is marked with a value k, we say that p is marked with k *tokens*. Weights may be assigned to the edges of PN, however, in this paper we use only the ordinary PN where weights of the arcs are always equal to 1. Moreover, we allow a marking with only one token for each place.

Definition 1 (Murata 1989) A Petri net (PN) is a 5-tuple, $PN = (P, T, F, M_0, I)$ where
$P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places,
$T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of transitions,
$F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs,
$M_0 = P \rightarrow \{0, 1, 2, 3, \ldots\}$ is the initial marking,
$I = (F' \subseteq (P \times T)) \rightarrow \{0, 1\}$ where 1 represents a regular arc and 0 an inhibitor arc, and
$P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.   □

We use $m(p)$ to denote the marking (or token) of place p, $i(p, t) = 0$ to denote an inhibitor arc and $i(p, t) = 1$ to denote a regular arc.

A transition (place) has a certain number (possibly zero) of input and output places (transitions).

Definition 2 (Murata 1989) Given a PN, the input and output set of transitions (places) for each place p_i (t_i) are defined as,

the set of input transitions of p_i, denoted $\bullet p_i = \{t_j | (t_j, p_i) \in F\}$

the set of output transitions of p_i, denoted $p_i \bullet = \{t_j | (p_i, t_j) \in F\}$, and

the input and output set of places for each transition t_i are defined as,

the set of input places of t_i, denoted $\bullet t_i = \{p_j | (p_j, t_i) \in F\}$

the set of output places of t_i, denoted $t_i \bullet = \{p_j | (t_i, p_j) \in F\}$. □

At any time a transition is either *enabled* or *disabled*. A transition t_i is enabled if each place in its input set $\bullet t_i$ has at least one token (in case of an inhibitor arc, t_i is enabled if there is no token in that input place). An enabled transition can fire. In order to simulate the dynamic behavior of a system, a marking in a PN is changed when a transition fires. Firing of t_i removes the token from each place in $\bullet t_i$ (no token is removed in case of an inhibitor arc), and deposits it into each place in $t_i \bullet$. The movement of tokens has been depicted in figure 2. The consequence of firing a transition results in a change from the original marking M to a new marking M'. For the sake of simplicity, we assume firing of a transition is an instantaneous event. The firing rules can be formally stated as follows:

Definition 3

1. A transition t_i is said to be enabled if $\forall p_j \in \bullet t_i$, either $(m(p_j) > 0) \wedge (i(p_j, t_i) = 1)$ or $(m(p_j) \not> 0) \wedge (i(p_j, t_i) = 0)$;
2. Firing an enabled transition t_i results in a new marking M' as follows: $\forall p_j \in \bullet t_i$, and $\forall p_k \in t_i \bullet$, if $m(p_j) > 0$ then $m'(p_j) = m(p_j) - 1 \wedge m'(p_k) = m(p_k) + 1$, otherwise $m'(p_k) = m(p_k) + 1$. □

Example 3 Figure 2 shows an example of a simple PN. It comprises of four places p_1, p_2, p_3, and p_4, and two transitions t_1 and t_2. The input and output sets of the places and transitions are as follows: $\bullet t_1 = \{p_1, p_2\}$, $\bullet t_2 = \{p_2\}$, $t_1 \bullet = \{p_3\}$, $t_2 \bullet = \{p_4\}$, $\bullet p_3 = \{t_1\}$, $\bullet p_4 = \{t_2\}$, $p_1 \bullet = \{t_1\}$, and $p_2 \bullet = \{t_1, t_2\}$. Note that the arc from p_2 to t_2 is an inhibitor arc.
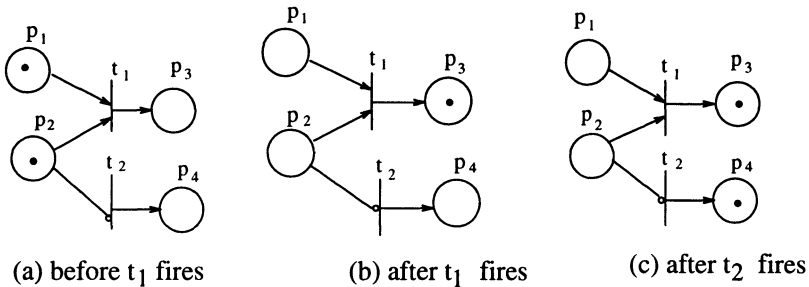

(a) before t_1 fires   (b) after t_1 fires   (c) after t_2 fires

Figure 2 An example of PN

The initial state of the PN is shown in figure 2(a) where p_1 and p_2 are both marked with one token each. Since both places in the input set of t_1 are marked (i.e., both $m(p_1)$, $m(p_2) > 0$) and the arcs from places p_1 and p_2 to t_1 are regular arcs (i.e., $i(p_1, t_1) = i(p_2, t_1) = 1$) t_1 is enabled. However, t_2 is not enabled because the arc from p_2 to t_1 is an inhibitor arc and p_2 is marked

(i.e., $m(p_2) \neq 0$). After t_1 fires it results in a new marking where the tokens from p_1 and p_2 are removed and a token is placed in p_3, as shown in figure 2(b). Since p_2 becomes empty after the firing of t_1, it now enables t_2 (because the arc from p_2 to t_2 is an inhibitor arc), but disables t_1. As a result, t_2 fires and places a token in p_4, as depicted in figure 2(c). Since there are no more transitions to fire, the PN stops (said to be not *live*) and thus the PN in figure 2(c) is the final marking.   □

Definition 4 A marking M is said to be *reachable* from a marking M_0 if there exists a sequence of firings that transforms M_0 to M.   □

Reachability is a fundamental property for studying the dynamic properties of any system. It has been shown (Kosaraju 1982) that the reachability problem is decidable although it takes at least exponential space and time.

5   PETRI NET REPRESENTATION OF MLS WORKFLOW TRANSACTIONS

In this section, first we will show how the various types of control-flow dependencies can be modeled using PNs. Then we will extend PNs to incorporate security levels.

A task in its simplest form consists of a set of states and a set of transitions that changes the state of the task from one state to the another. Let the initial state of a task tw_i be in_i, execution state be ex_i, commit state be cm_i and abort state be ab_i. Transition *begin* (b_i) moves the task from in_i to ex_i, transition *commit* (c_i) moves tw_i from ex_i to cm_i and transition *abort* (a_i) moves tw_i from ex_i to ab_i. PN representation of tw_i is shown in figure 3.
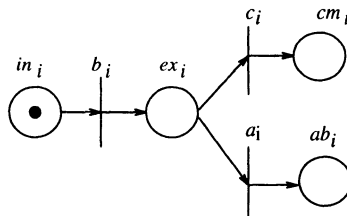

Figure 3 A PN representation of task tw_i

5.1   PN Representation of Control Flow Dependencies

In this section, we will show how a control flow dependency can be modeled as a PN. First we present the PN representation of a general control flow dependency and then show each type of control-flow dependency discussed in section 2.1 as a PN.

A control flow dependency, in general, is as follows: Given any two tasks tw_i and tw_j, tw_j can enter state st_j only after task tw_i enters state st_i. This can be explicitly represented by the PN

shown in figure 4. We add a buffer state $b_{ij}x$ which has an input arc from $\bullet st_i$ (or t_i in figure 4) and an output arc from $b_{ij}x$ to $\bullet st_j$ (or t_j in figure 4).

Figure 4 shows the PN model for the general control flow dependency. In this figure, and all the subsequent figures, we use thick lines to represent the portion of the net included to enforce the dependency. Here, tasks tw_i and tw_j move from states st_{i-1} and st_{j-1} to st_i and st_j when the events t_i and t_j occur (or transitions t_i and t_j fire), respectively. Let us inspect how the control flow dependency can be enforced with this PN model.

Before task tw_i enters state st_i (or before the firing of t_i), $b_{ij}x$ is not marked, and therefore prevents transition t_j to fire (or task tw_j to enter st_j) by disabling it. However, when task tw_i enters state st_i with the firing of transition t_i, one token is deposited in both st_i and $b_{ij}x$. At this point, transition t_j fires because both of its input places are marked. Thus, task tw_j is allowed to enter state st_j only if task tw_i enters state st_i.

There are two reasons for using a buffer state $b_{ij}x$ to connect their preceding transitions t_i and t_j instead of directly connecting st_i to st_j via a transition. (1) It ensures that once tw_i enters st_i, tw_j can enter st_j whenever tw_j is ready to enter st_j (t_j is enabled). On the other hand, using a transition to directly connect st_i to st_j, gives a different interpretation that tw_j *must* enter st_j when tw_i enters st_i, which is an incorrect representation of the original dependency. (2) It allows modeling of multiple dependencies (e.g. $tw_i \xrightarrow{bc} tw_j$ and $tw_i \xrightarrow{a} tw_j$).
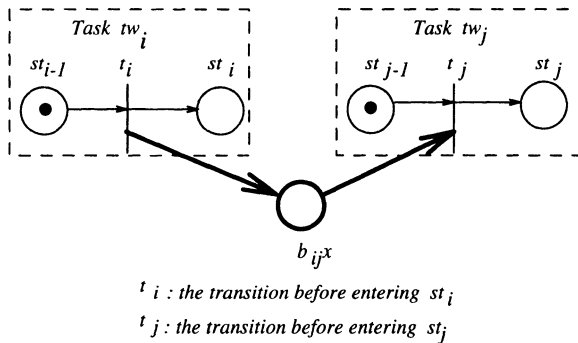

t_i : the transition before entering st_i

t_j : the transition before entering st_j

Figure 4 A PN model for Control-flow dependency

Although the PN representation of various types of control flow dependencies can be constructed from the general case shown in figure 4, some types need modification. In the following, we will present the PN representation of the type of dependencies presented in section 2.

Strong commit dependency: To model this dependency, we insert an additional buffer state $b_{ij}c$ and connect an incoming arc from c_i to $b_{ij}c$ and another outgoing arc from $b_{ij}c$ to c_j as shown in figure 5. A token is placed in $b_{ij}c$ only when tw_i commits. So task tw_j will not be allowed to commit until $\bullet c_j$ as well as $b_{ij}c$ are marked.

Abort dependency: Here we make an assumption that once tw_j commits before tw_i aborts, it cannot be aborted later. We first create a buffer state $b_{ij}a$ where we connect an incoming arc from a_i and insert an inhibitor arc to every transition except a_j (i.e., to b_j and c_j), as shown in figure 6. Once task tw_i aborts, these two inhibitor arcs prevent (1) the starting of execution of
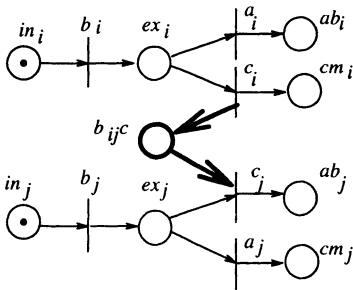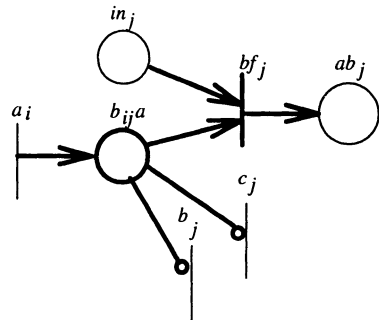
Figure 5 A PN for modeling strong commit dependency

Figure 6 A PN for modeling an abort dependency

tw_j if it has not already started by using a buffer transition bf_j such that $\bullet bf_j = \{in_j, b_{ij}a\}$ and $bf_j\bullet = \{ab_j\}$ (This ensures that tokens in in_j and $b_{ij}a$ will fire bf_j, thereby moving tw_j to state ab_j.), and (2) the commit of tw_j if it has not yet committed. When a_i fires, $b_{ij}a$ is filled with a token, thus both b_j and c_j cannot fire, but only a_j can fire.
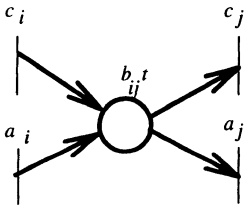
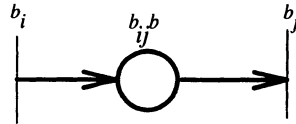**Figure 7** A PN model for termination dependency

Figure 8 A PN model for begin dependency

Termination dependency: To represent this dependency, we insert a buffer state $b_{ij}t$ such that $\bullet b_{ij}t = \{c_i, a_i\}$ and $b_{ij}t\bullet = \{c_j, a_j\}$ as shown in figure 7. This PN ensures that only when task tw_i terminates by firing either c_i or a_i, $b_{ij}t$ will be marked and thus allows c_j or a_j to be enabled.

Begin dependency: Similar to the modeling of strong commit dependency, we add a buffer state $b_{ij}b$ such that $\bullet b_{ij}b = \{b_i\}$ and $b_{ij}b\bullet = \{b_j\}$ as shown in figure 8.

Begin-on-commit dependency: This is similar to begin dependency except that we replace $b_{ij}b$ with $b_{ij}bc$ and connect arcs from c_i to b_j as in figure 9.

Group commit: Group commit must ensure that either both tw_i and tw_j commit or neither of them commit. To illustrate this using PN, we create two buffer states $b_{ij}gc$ and $b_{ji}gc$ such that
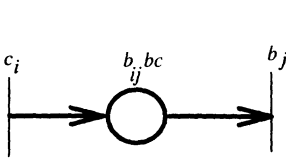
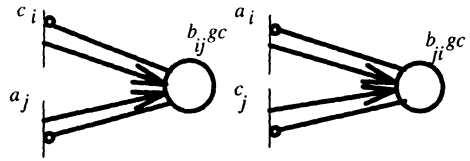**Figure 9** A PN model for begin-on-commit dependency

Figure 10 A PN model for group-commit dependency

$\bullet b_{ij}gc = b_{ij}gc\bullet = \{c_i, a_j\}$ and $\bullet b_{ji}gc = b_{ji}gc\bullet = \{a_i, c_j\}$ as shown in figure 10. By modeling this way, we ensure that once c_i (or a_i) fires, a_j (or c_j) will be disabled and thus only c_j (or a_j) is enabled. Similar argument can be made if c_j (or a_j) fires first.

Control-flow dependency with data flow: The PN model of the general control-flow dependency with data flow is similar to figure 4, except that the token is associated with a data value (represented as a shaded dot instead of a filled dot as shown in figure 11). This represents that task tw_j can enter state st_j only after tw_i enters state st_i and tw_i passes a value of data contained in the token to tw_j. To derive the PN representation of a specific control flow with data flow, one can combine this PN with the PN for that specific type of dependency.
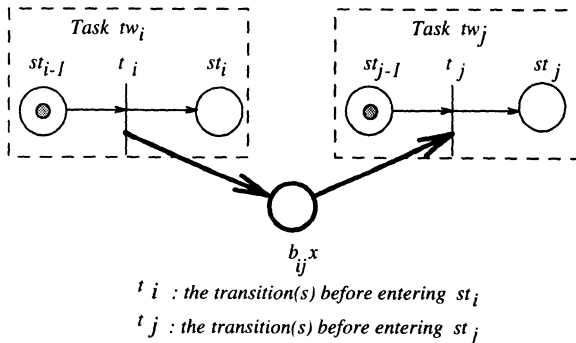

t_i : the transition(s) before entering st_i

t_j : the transition(s) before entering st_j

Figure 11 A PN model for control flow dependency with data flow

5.2 A Secure Petri Net (SPN) Model

To model MLS workflows, we extend the ordinary PN by incorporating multilevel secure constraints, which we call *Secure Petri Net* (SPN). We associate security level to each place as well as to each token, which results in various types of places and tokens. The idea is similar to assigning strong types to places as in typed Petri nets and assigning colors to tokens as in colored Petri nets (Peterson 1981), however, we incorporate both these techniques into SPN.

Let S be a partially ordered set of security levels. Each task tw_i is assigned a security level

such that $s(tw_i) = s \in S$. All places and transitions within a task assume the same security level of the task. We use $s(p_i)$, where $s \in S$ to represent the security level associated to place p_i.

To distinguish the typed places from the regular places, we use double circles as opposed to single circles. Similarly, the token is represented as a "shaded dot" instead of a "filled dot." We use the same notation as in case of control- flow dependency with data flow. The reason is as follows. Assigning a security level to a token and assigning a value to a token are analogous. Therefore, although we do not explicitly address value dependencies in this paper, we believe that they can be modeled in a similar way as inter-level dependencies.

The extended Petri net is defined as follows:

Definition 5 A *Secure Petri Net* (SPN) is a 2-tuple, $SPN = (PN, S)$, such that $s(p_i)$, where $s \in S$ is the security level of place p_i, and $s(m(p_i))$, where $s \in S$ is the security level of the token in p_i □

The following security constraint restricts the security level of tokens.

● A token $m(p_i)$ is allowed to reside in place p_i only if $s(m(p_i)) = s(p_i)$.

The new firing rules for the SPN with typed tokens and places are as follows:

Definition 6

1. A transition t_i is said to be enabled if $\forall p_j \in \bullet t_i$, either $(m(p_j) > 0) \wedge (i(p_j, t_i) = 1)$ or $(m(p_j) \not> 0) \wedge (i(p_j, t_i) = 0)$ (same as rule 1 in definition 3)
2. Firing an enabled transition t_i results in a new marking M' as follows: $\forall p_j \in \bullet t_i$ and $\forall p_k \in t_i \bullet$,

    (a) if $s(m(p_j)) > s(p_k)$ then $m'(p_j) = m(p_j) - 1$,
    (b) if $s(m(p_j)) \leq s(p_k)$ then
        $(m'(p_j) = m(p_j) - 1 \wedge m'(p_k) = m(p_k) + 1)$, whenever $(i(p_j, t_i) = 1)$
        $m'(p_k) = m(p_k) + 1$, whenever $(i(p_j, t_i) = 0)$.
        In either case, $s(m'(p_k)) = s(p_k)$.   □

The second firing rule states that upon firing, one token is reduced from each place p_i in $\bullet t_i$ and one token is inserted to each place p_k in the output set of t_i according to the two rules: (1) if the original token in p_i (i.e. $m(p_i)$) has a security level higher than that of p_k, upon firing, we only remove tokens from p_i but do not insert any token to p_k. (2) if the original token in p_i has a security level equal to or lower than that of p_k, upon firing, we remove a token from p_i and insert a token to p_k. The security level of the inserted token will be equal to the level of the place where the token is being inserted. This does not introduce any covert channels because according to our firing rules, no token is allowed to pass from a place with higher security level to a place with lower security level.

Example 4 As an example, consider the SPN shown in figure 12, where transition t_1 has three input places p_1, p_2 and p_3 with security levels *high*, *high* and *very high*, respectively, and one output place p_4 with $s(p_4) = high$. Assume $m(p_2) = m(p_3) = 1$ and $m(p_1) = 0$, and while arcs

(p_2, t_1) and (p_3, t_1) are regular arcs, (p_1, t_1) is an inhibitor arc. The security levels of the tokens are as follows: $s(m(p_2)) = high$ and $s(m(p_3)) = very\ high$.

When t_1 fires, tokens are removed from both p_2 and p_3, and a token is placed in p_4 since there is a token $m(p_2)$ whose security level is less than or equal to that of p_4. The security level assigned to this inserted token is *high* because $s(p_4) = high$. □

(a) before t_1 fires   (b) after t_1 fires
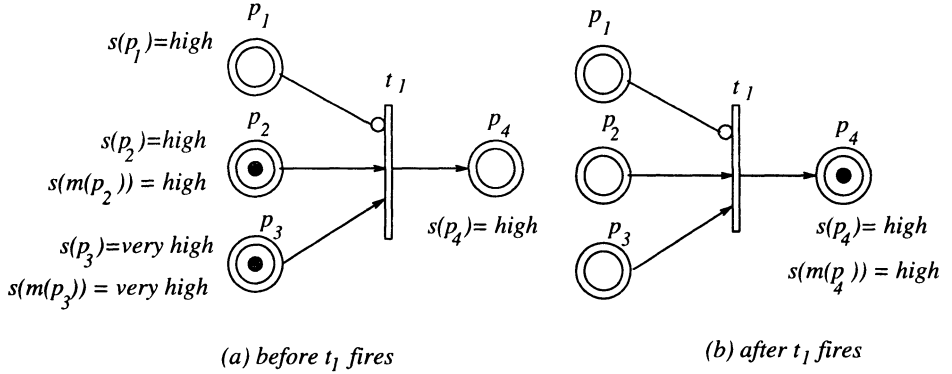
Figure 12 An example depicting the behavior of a Secure Petri Net

5.3 Modeling inter-level dependencies

Now we propose a mechanism that can automatically detect and prevent covert channels by disabling all *high-to-low* flow for every control flow dependency $tw_i \xrightarrow{x} tw_j$. The buffer place b_{ij} is assigned the security level of tw_j (i.e., $s(b_{ij}x) = s(tw_j)$). We create two buffer places $b_{ij}x1$ and $b_{ij}x2$ such that $s(b_{ij}x1) = s(tw_i)$ and $s(b_{ij}x2) = s(tw_j)$. Both these places are connected to $prevent_{ij}x$ with inhibitor arcs as figure 13 shows. This mechanism is required only for those dependencies without any inhibitor arcs such as $tw_i \xrightarrow{c} tw_j$, $tw_i \xrightarrow{b} tw_j$, $tw_i \xrightarrow{bc} tw_j$, and $tw_i \xrightarrow{t} tw_j$.

This works as follows: $b_{ij}x1$ is initially marked with a token such that $s(m(b_{ij}x1) = s(b_{ij}x1))$. So transition $detect_{ij}x$ is enabled initially. For all cases where $(s(tw_i) \leq s(tw_j))$, i.e., *low-to-high*, or dependencies among tasks of the same level, the token $m(b_{ij}x1)$ will be moved from $b_{ij}x1$ to $b_{ij}x2$. Since $b_{ij}x2$ is not empty, $prevent_{ij}x$ is not enabled. Thus the specified control flow dependency is enforced. If $(s(tw_i) \not\leq s(tw_j))$, (i.e., to include dependencies between *high-to-low* and between incomparable) no token will be inserted into $b_{ij}x$ when t_i fires.

The transition $detect_{ij}x$ fires but token $m(b_{ij}x1)$ will be removed from $b_{ij}x1$ but will not be inserted into $b_{ij}x2$ because $s(m(b_{ij}x1) = s(b_{ij}x1) \not\leq s(b_{ij}x2)$. These two empty places will further enable the transition $prevent_{ij}x$. $prevent_{ij}x$ fires and inserts a token with the security level $s(m(b_{ij}x)) = s(b_{ij}x) = s(tw_j)$ to its output place $b_{ij}x$. Because $b_{ij}x$ is not filled with a token through the firing of t_i, but via the firing of $prevent_{ij}x$, the *high-to-low* dependency is not enforced and tw_j is processed independently. Thus, even though $s(tw_i) > s(b_{ij})$, it does not cause any covert channel.

For those dependencies with inhibitor arcs such as $(tw_i \xrightarrow{a} tw_j)$ and $(tw_i \xrightarrow{gc} tw_j)$, the dependency is already secure and thus they do not require this mechanism. As an example, consider a *high-to-low* abort dependency (refer to figure 6). The absence of token in buffer $b_{ij}a$ enables both b_j and c_j, which implies the dependency is no longer enforced.
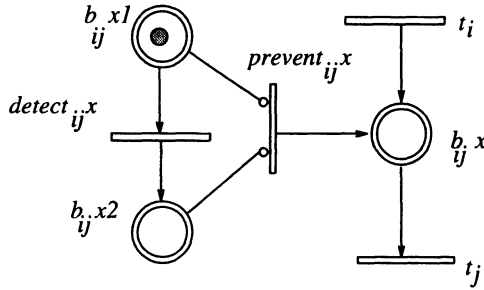

Figure 13 A PN model to enforce inter-level dependency

6   EXECUTION OF WORKFLOWS

Execution of a workflow transaction involves submitting tasks to the workflow management system (WFMS) while ensuring all the task dependencies being preserved. We will show how this can be accomplished using PNs. It is important to note that these algorithms can be used even to execute concurrent workflows with little modification. Our model is helpful when scheduling workflow transactions concurrently, especially when transactions are *ad hoc* in nature. The SPN of each newly submitted transaction can simply be added to the existing SPN of the currently executing workflows.

Algorithm 1 [An Algorithm to Construct $SPN(TW)$]

for each $tw_i \in TW$,
    /* construct PN of tw_i as follows:*/
    create places $P_i = \{in_i, ex_i, cm_i, ab_i\}$
    create transitions $T_i = \{b_i, c_i, a_i\}$
    connect P_i and T_i with directed arcs as follows:
    $\bullet in_i = \emptyset, in_i \bullet = \{b_i\}, \bullet b_i = \{in_i\}, b_i \bullet = \{ex_i\}, \bullet \bullet ex_i = \{b_i\}, ex_i \bullet = \{c_i, a_i\},$
    $\bullet c_i = \{ex_i\}, c_i \bullet = \{cm_i\}, \bullet a_i = \{ex_i\}, a_i \bullet = \{ab_i\}, \bullet cm_i = \{c_i\},$
    $cm_i \bullet = \emptyset, \bullet ab_i = \{a_i\}, ab_i \bullet = \emptyset$
end{for}

for each $tw_i \xrightarrow{x} tw_j \in TW$, where tw_j can enter state st_j only when tw_i enters state st_i

1. add the buffer place $b_{ij}x$ and arcs as shown in figure 4$^{\|}$
2. /*Construct $SPN(TW)$ as follows: */
   create two places $b_{ij}x1, b_{ij}x2$ and two transitions $detect_{ij}x, prevent_{ij}x$
   connect arcs such that $\bullet b_{ij}x1 = \emptyset$, $b_{ij}x1\bullet = \{detect_{ij}x, prevent_{ij}x\}$ where $i(b_{ij}x1, prevent_{ij}x)$
   $= 0^{**}$
   $\bullet b_{ij}x2 = \{detect_{ij}x\}$, $b_{ij}x2\bullet = \{prevent_{ij}x\}$ where $i(b_{ij}x2, prevent_{ij}x) = 0$,
   $prevent_{ij}x\bullet = \{b_{ij}x\}$
3. /*assign the security level to the places as follows:*/
   $s(in_i), s(ex_i), s(cm_i), s(ab_i) \leftarrow s(tw_i),$
   $s(in_j), s(ex_j), s(cm_j), s(ab_j) \leftarrow s(tw_j),$
   $s(b_{ij}x) \leftarrow s(tw_i), s(b_{ij}x1) \leftarrow s(tw_i), s(b_{ij}x2) \leftarrow s(tw_j)$

end{for}

Algorithm 2 [An Algorithm to Execute TW]

/* Mark SPN with M as follows: */
for each $tw_i, tw_j \in TW$
    $m(in_i) \leftarrow 1$, $m(in_j) \leftarrow 1$ and $m(b_{ij}x1) \leftarrow 1$
    /*assign security level to tokens as follows:*/
    $s(m(in_i)) \leftarrow s(in_i);$
    $s(m(in_j)) \leftarrow s(in_j), s(m(b_{ij}x1)) \leftarrow s(b_{ij}x1),$
end{for} $M' \leftarrow \emptyset$ /* initialize next state of marking */
$M^{-'} \leftarrow \emptyset$ /* initialize previous state of marking */
while$M^{-'} \neq M$
    /* execute PN by firing the enabled transition */
    for each $t_i \in SPN(TW)$
        $\forall p_j, p_k \in SPN(TW)$, where $p_j \in \bullet t_i$ and $p_k \in t_i\bullet,$
            1. if $s(m(p_j)) > s(p_k)$ then $m'(p_j) = m(p_j) - 1,$
            2. if $s(m(p_j)) \leq s(p_k)$ then
                $(m'(p_j) = m(p_j) - 1 \wedge m'(p_k) = m(p_k) + 1)$, whenever $(i(p_j, t_i) = 1)$
                $m'(p_k) = m(p_k) + 1$, whenever $(i(p_j, t_i) = 0)$
                $s(m'(p_k)) = s(p_k)$
    $M^{-'} \leftarrow M$
    $M \leftarrow M'$
end{while}

As an example, algorithm 1 can be used to construct SPN for the workflow transaction in example 1, which is shown in figure 14. The SPN thus constructed can be executed using algorithm

---

$^{\|}$For simplicity, we consider only the general case of control flow dependency. For a specific control flow dependency, buffer transitions and additional buffer places may be required as presented in section 5.1.
**We omit explicitly specifying $i(b, t)$ if its value is equal to 1.

2, which is as follows. First all the places representing the initial state of the task (in_i) and the buffer state $b_{ij}x1$ of every task dependency $tw_i \xrightarrow{x} tw_j$ in TW are marked with a token. This forms the initial marking M. When all the enabled transitions fire, it results in a new marking M'. This continues until no new transitions fire. Notice that in figure 14, the *high-to-low* dependency $tw_2 \xrightarrow{bc} tw_3$ is detected and disabled because a token is deposited in $b_{23}bc$ by firing $prevent_{23}bc$. This immediately enables b_3 without waiting for the commit of tw_2 thereby eliminating covert channels.
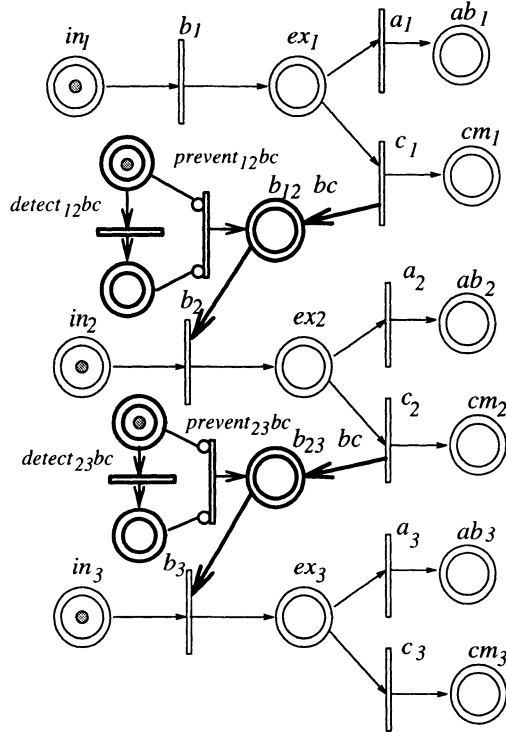

Figure 14 SPN of workflow transaction in example 1

7   CONCLUSIONS AND FUTURE RESEARCH

In this paper, first we have presented a multilevel secure workflow transaction model where we identify the task dependencies that have to be prevented in order to eliminate covert channels. Then we have used Petri nets to model various types of control flow dependencies and extended the traditional PN to SPN which automatically detects and prevents all task dependencies that

violate security. We have also proposed algorithms to construct an SPN for a given MLS workflow transaction and to execute them. Note however preventing all *high-to-low* dependencies may result in an incorrect workflow execution.

As part of future work, we intend to represent value dependencies and external dependencies as PNs. Representation of external dependencies involve using a special case of PNs known as *timed PNs*. We intend to implement and perform reachability analysis of SPN.

ACKNOWLEDGMENT

REFERENCES

Bell, D. & LaPadula, L. (1976), Secure computer systems: Unified exposition and multics interpretation., Technical Report MTR-2997, The Mitre Corporation, Bedford, MA.

Biliris, A., Dar, S., Gehani, N., Jagadish, H. & Ramamritham, K. (1994), ASSET: a system for supporting extended transactions, *in* 'Proc. ACM SIGMOD Int'l. Conf. on Management of Data', Minneapolis, MN, pp. 44–54.

Blaustein, B. T., Jajodia, S., McCollum, C. D. & Notargiacomo, L. (1993), A model of atomicity for multilevel transactions, *in* 'Proc. IEEE Symposium on Security and Privacy', Oakland, California, pp. 120–134.

Chrysanthis, P. (1991), ACTA, A framework for modeling and reasoning about extended transactions, PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst.

Denning, D. E. (1982), *Cryptography and Data Security*, Addison-Wesley, Reading, MA.

Elmagarmid, A. K. (1992), *Database Transaction Models for Advanced Applications*, Morgan Kaufmann, San Mateo. California.

Elmagarmid, A. K., Leu. Y., Litwin, W. & Rusinkiewicz, M. (1990), A Multidatabase Transaction Model for InterBase, *in* 'Proc. 16th Int'l. Conf. on Very Large Data Bases', Briabane, Australia, pp. 507–518.

Georgakopoulos, D., Hornick, M. & Sheth, A. (1995), 'An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure', *Distributed and Parallel Databases* pp. 119–153.

Georgakopoulos, D. et al. (1993), 'An Extended Transaction Environment for Workflows in Distributed Object Computing', *Bulletin of IEEE Technical Committee on Data Engineering* **16**(2), 24–27.

Kosaraju, S. R. (1982), Decidability and reachability in vector addition systems, *in* 'Proc. of the 14th ACM Symposium on Theory of Computing', pp. 267–281.

Murata, T. (1989), 'Petri nets: Properties, analysis and applications', *Proceedings of the IEEE* **77**(4), 541–580.

Peterson, J. L. (1981), *Petri net theory and modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ.

Rusinkiewicz, M. & Sheth, A. (1994), Specification and Execution of Transactional Workflows, *in*
     W. Kim, ed., 'Modern Database Systems: The Object Model, Interoperability, and Beyond',
     Addison-Wesley.
Sheth, A., Rusinkiewicz, M. & Karabatis, G. (1993), 'Using Polytransactions to Manage Interde-
     pendent Data', *Bulletin of IEEE Technical Committee on Data Engineering* 16(2), 37–40.

BIOGRAPHY

Vijayalakshmi Atluri is an Assistant Professor of Computer Information Systems in the MS/CIS
Department at Rutgers University. She received her B.Tech. in Electronics and Communications
Engineering from Jawaharlal Nehru Technological University, Kakinada, India, in 1977, M.Tech.
in Electronics and Communications Engineering from Indian Institute of Technology, Kharagpur,
India, in 1979, and Ph.D. in Information Technology from George Mason University, USA, in
1994. Her research interests include Information Systems Security, Database Management Sys-
tems, Workflow Management and Distributed Systems.

Wei-Kuang Huang received his B.S. in Naval Architecture Engineering from National Taiwan
University, Taiwan in 1987, and M.S. in Management Information System from Boston University,
Massachusetts in 1991. He is currently a Ph.D. candidate in the MS/CIS department at Rutgers
University, New Jersey. His research is in the areas of database security, workflows management
and Petri net modeling and analysis.