

# Analyzing Separation of Duties in Petri Net Workflows

Konstantin Knorr and Harald Weidner

Department of Information Technology  
University of Zurich  
CH-8057 Zurich  
{knorr,weidner}@ifi.unizh.ch

**Abstract.** With the rise of global networks like the Internet the importance of workflow systems is growing. However, security questions in such environments often only address secure communication. Another important topic that is often ignored is the separation of duties to prevent fraud within an organization. This paper introduces a model for separation of duties in workflows that have been specified with Petri nets. Rules will be given as facts of a logic program and expressed in propositional logic. The program allows for simulating and analyzing workflows and their security rules during build time.

**Keywords.** Logical programming, Petri net, separation of duties, workflow

## 1 Introduction

It is well known that many computer related criminal activities are performed by insiders (Anonymous 1985, CSI 1999). One of the most prominent threats is fraud that is particularly difficult to detect in computerized environments such as workflow systems. Therefore it is of great importance to implement mechanisms to prevent such illegal activities. Separation of duties (SoD) has been identified by many authors as an efficient mechanism to prevent fraud within organizations (Ahn and Sandhu 1999, Bertino et al. 1999, Bussler 1995, Sandhu 1990, Stormer et al. 2000). It is in particular useful when applied to dynamic processes such as workflows. The physical and logical separation of tasks and their subjects can improve the prevention of fraudulent activities.

We present in this paper a simple logical representation for SoD rules that will restrict the execution of tasks by subjects, thus implementing a dynamic separation of duties. The SoD analysis proposed will be done in two steps: in a first step, logical programming (Prolog) will be used to calculate all valid execution chains of a workflow. In a second step, the result of the first step is used for further analysis (e.g. workload). A Petri net (PN) will express syntax and semantics of the business processes under consideration. The formal approach has the advantage of being analyzable before implementing a given workflow in a business environment. The specification

can be searched for potential flaws like deadlocks, non-reachable end-states, and contradicting SoD rules.

The remainder of the paper has the following structure: Section 2 gives background information on workflows, roles, SoD, PNs, and PN workflows. Section 3 introduces an example that is used for illustration purposes throughout the paper. A formal SoD model is described in Section 4. Section 5 uses logical programming to analyze SoD rules of a business process. Section 6 discusses the findings of the paper, contrasts them to related work, and mentions further research topics.

## 2 Background

### 2.1 Workflows

Workflow management is an essential research area in computer science. It is an emerging technology used to automate and ‘streamline’ frequent business processes. This approach is especially useful for processes that rely on electronic documents because the workflow management system (WfMS) can administer and process the data objects. A workflow is an executable and computer-understandable business process whose administration, modeling, and execution is supported by a software system called WfMS. Before a workflow can be executed, it has to be described in a way, the WfMS is able to understand. This description is called a *workflow specification*. The definition is made during *build time*, i.e. before a workflow can be executed. During *run time* of the system many instances of the workflow are generated according to the specification. The main elements of a workflow specification are

- tasks,
- subjects,
- roles,
- and the control flow.

A workflow consists of several tasks whose chronological and logical order is given by the control flow. To describe a task, it has to be specified which subjects/roles are allowed to execute the task. Subjects can be associated with persons but also with machines and computer programs. For more comprehensive information see Cichocki et al. 1998 and Georgakopoulos et al. 1995.

### 2.2 Roles

In a workflow environment, usually tasks are not linked directly to subjects. The concept of *roles* forms a middle layer between subjects and tasks. Roles allow for tying the execution of a task to a specific group of persons with specific skills or qualifications. When a representative of a role leaves the company, task definitions can remain unchanged. Just the role has to be ‘untied’ from that person (Lawrence 1993).

An important field of applying the role concept is security. Access rights are enforced on roles and not on subjects which simplifies and cheapens security administration (cf. Proceedings of the 5<sup>th</sup> ACM Workshop on Role-based Access Control).

### 2.3 Separation of Duties

“SoD is a policy to ensure that failures of omission or commission within an organization are caused only by collusion among individuals and, therefore, are riskier and less likely, and that chances of collusion are minimized by assigning individuals of different skills or divergent interests to separate tasks” (Gligor et al. 1998).

SoD therefore is an important security mechanism to prevent fraud within an organization. Clark and Wilson (1987) stress the importance of SoD mechanisms in commercial settings. Within a business process context SoD rules express task dependencies and should be part of a company’s security policy. Only recently the combination of workflow management and SoD has found considerable interest in the research community (Bussler 1995, Bertino et al. 1998, Stormer et al. 2000). In a workflow context, SoD has to be divided and extended into static SoD (SSoD) and dynamic SoD (DSoD). SSoD enforces certain rules during build time of the workflow. SSoD rules are therefore part of the workflow specification. In contrast, DSoD is enforced during run time. Examples: In a workflow different roles are specified for different tasks (SSoD). In a travel expenses workflow a manager should not be allowed to apply for the reimbursement of (task 1) and approve (task 2) his own travel expenses (DSoD). For the remainder of this paper we will concentrate on DSoD and call it SoD if not otherwise stated.

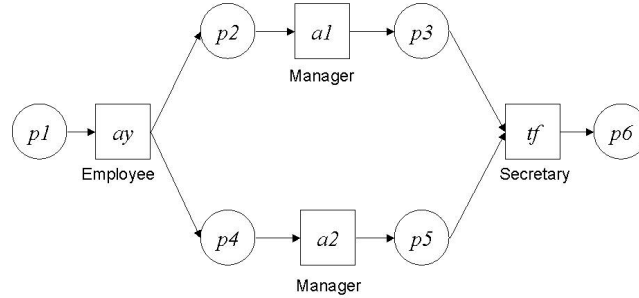
### 2.4 Petri nets

Petri nets (PN) originated with Carl Adam Petri (Petri 1962). Nowadays, their theory and application is a vast research area with many publications. This section gives the basic definitions of Place/Transition Nets with arc weights equal to 1 (Reisig 1985).

**Definition 1 (Petri net)** A Petri net  $N$  is a triple  $N = (P, T, F)$ .  $P$  is the finite set of the *places*,  $T$  the finite set of the *transitions* with  $P \cap T = \emptyset$ . The *flow relation*  $F$  is defined by  $F \subseteq (P \times T) \cup (T \times P)$ . Let  $y \in P \times T$ .  $\bullet y$  is called the *preset* of  $y$  and is defined by  $\bullet y := \{x \in P \cup T \mid (x, y) \in F\}$ .  $y\bullet$  is called the *postset* of  $y$  and is defined by  $y\bullet := \{x \in P \cup T \mid (y, x) \in F\}$ . An element of  $\bullet y$  and  $y\bullet$  is called an *input place/transition* and *output place/transition*, respectively.

Figure 1 shows an example of a Petri net consisting of the places  $p_1, \dots, p_6$  and the transitions  $ay, a1, a2, tf$ . The flow relation  $F$  equals to  $F = \{(p_1, ay), (ay, p_2), (ay, p4), (p2, a1), (a1, p3), (p3, tf), (p4, a2), (a2, p5), (p5, tf), (tf, p6)\}$ . The graphical interpretation of a Petri net is a bipartite graph, i.e. places can only be connected to transitions, transitions can only be connected to places. Places are represented graphically as circles, transitions as rectangles. The graphical interpretation of an element  $(x,$

y)  $F$  is an arrow from  $x$  to  $y$ . The preset and postset of a transition are sets of places — possibly empty. The preset and postset of a place are sets of transitions — possibly empty, too. Some presets and postsets from the example are  $\bullet tf = \{p3, p5\}$ ,  $\bullet p_1 =$ , and  $tf\bullet = \{p6\}$ .



**Figure 1. Example of a Petri net**

**Definition 2 (Behavior of Petri nets)** A marking  $M$  is a mapping  $M: P \rightarrow N_0$  that associates with each place a number of *tokens*. A transition  $t$  is called *activated* under marking  $M$ , if  $M(p) > 0$  for all  $p \in \bullet t$ . An activated transition can *fire*. If a transition  $t$  fires,  $M$  changes to  $M^*$  such that  $M^*(p) = M(p) - (p, t) + (t, p)$  for every place of the net, where  $(x, y) = 1$  if  $(x, y) \in F$  and  $(x, y) = 0$  otherwise.

The behavior of Petri nets shall be illustrated following the example in Figure 1. Let the start marking be  $M(p1) = 1$  and  $M(p) = 0$  for all other places. A token is put on place  $p1$ . Transition  $a1$  is activated. If  $a1$  fires, the token from  $p1$  duplicates and moves to  $p2$  and  $p4$ . Transitions  $a1$  and  $a2$  are activated now. If  $a1$  fires, the token from  $p2$  moves to  $p3$ . Similarly, if  $a2$  fires, the token from  $p4$  moves to  $p5$ .  $tf$  can fire now, merging the tokens from  $p3$  and  $p5$  into a single token in  $p6$ . Now, there are no more activated transitions.

Definitions 1 and 2 give the ‘classical’ definitions for Petri nets. During the last years so called *High Level Petri nets* have been introduced. The major differences are that the tokens can be distinguished (Jensen 1992).

## 2.4 Petri net workflows

The fundamental idea behind the connection of PNs with workflows is to associate activities in the workflow with transitions in the PN. Through this association, the execution of an activity can be interpreted as the firing of a transition. The flow relation of the PN gives the control flow of the workflow. The marking of a PN can be associated with the state of the workflow. Start and end activities are derived from the start and end marking of the net. Subjects and roles are interpreted as attributes of transitions (Knorr 2000).

**Definition 3 (Petri net workflow)** A *Petri net workflow* is a workflow whose tasks and control flow are specified through a Petri net where  $T$  is the set of tasks/transitions. Additionally,  $R$  is the set of roles and  $S$  the set of subjects. Function  $f_{TR}: T \rightarrow 2^R$  assigns a set of roles to each task,  $f_{SR}: S \rightarrow 2^R$  a set of roles to each role<sup>1</sup>.

Petri net workflows offer the following advantages: The ‘correctness’ of their static and dynamic properties can be proven. There are various publications on this topic (Adam et al. 1998, Kindler and van der Aalst 1999, van der Aalst 1997). Note that it is possible to perform a validity check of the specification before run time, preventing any inconsistencies during execution resulting from an erroneous workflow specification. Next to their formal definition, PN workflows have an intuitive graphical interpretation that is well suited to express the dynamic nature of a workflow. Furthermore, PN could be used to standardize workflow specifications, an ongoing project led by the Workflow Management Coalition<sup>2</sup>.

### 3 Sample business process

The last section introduced several concepts on an abstract level. This section gives an example of a business process that will be used for illustration purposes throughout the remainder of the paper. The business process deals with travel expenses reimbursement. Figure 1 shows the associated PN workflow. The workflow consists of four tasks: in a first task ( $ay$ ), an employee applies for the reimbursement of his travel expenses by filling out an application form. Two managers have to approve this report (tasks  $a1$  and  $a2$ ). Finally, based on the approval of the managers a secretary will transfer the money to the employee's bank account ( $tf$ ). Note that an instance of this workflow is created for every travel of an employee.

Formally, the set  $T$  of all transitions is  $T=\{ay, a1, a2, tf\}$  and the set of roles  $R$  encompasses the manager, secretary and employee role, thus  $R=\{emp, man, sec\}$  if the roles are abbreviated. The role associated with each task in Figure 1 is written under the corresponding transition (e.g.  $f_{TR}(ay)=\{emp\}$ ). Let the subjects of the company be Carpenter, Butcher, Snyder, Fisher, and the brothers A. Smith and B. Smith, thus  $S = \{car, but, sny, fis, sma, smb\}$ . All six subjects are employees, Carpenter, Butcher and B. Smith are managers, Fisher and Snyder are secretaries (therefore  $f_{SR}(fis) = \{emp, sec\}$ ). Note that every manager (or secretary) is an employee, too. The partial order of roles builds up a so called *role hierarchy*.

Now we are ready to give SoD examples. The process definition in Figure 1 requires that different tasks are performed by different roles (the  $tf$  task by a secretary and the  $a1$  task by a manager). That corresponds to SSoD rules. However, DSoD is of greater interest in a workflow context. The following rules are reasonable for this specific business process:

<sup>1</sup> In this definition  $2^A$  denotes the power set of  $A$  – the set of all subsets of  $A$ .

<sup>2</sup> <http://www.wfmc.org>

- A manager should not be allowed to approve his/her own travel claim.
- B. Smith should not be allowed to approve the claim of his brother A. Smith.
- A secretary should not be allowed to transfer the refund of his/her own travel expenses.
- A manager should not be allowed to perform both approval tasks in the same workflow instance.

These examples clearly state the need for a formal model for SoD that takes into account the state of the underlying business process, which is the topic of the following section.

## 4 Formalizing separation of duties

This section introduces a SoD model based on PNs. For this purpose, the rules of a business process are stored in a *rule base* (RB).

**Definition 4 (SoD rule base)** RB is a set of entries (rules) of the form

$$(s1, t1) \quad \neg (s2, t2) \quad (1)$$

where  $(s1, t1), (s2, t2) \in S \times T$  ( $S$  is the set of subjects and  $T$  is the set of tasks). A rule says that if subject  $s1$  has done task  $t1$  then  $s2$  must not do  $t2$ .

Next to this separation of duties, a delegation of duties is also imaginable:  $(s1, t1)$   $(s2, t2)$ . If subject  $s$  has done a first task it might be reasonable that  $s$  performs a second task later on. Nevertheless, we restrict ourselves to separation since a delegation can be expressed by excluding all other subjects from performing the second task:

$$(s1, t1) \quad (s2, t2) \quad [ (s1, t1) \quad \neg (x, t2) \quad x \in S \setminus \{s2\} ] \quad (2)$$

This equivalence will generate a large number of SoD rules. For a discussion of the complexity of the SoD analysis see Section 6.

Not all rules given through (1) are meaningful. Therefore, the notion of *soundness* is introduced.

**Definition 5 (Soundness)** A rule in RB is sound if and only if

1.  $f_{TR}(t) \quad f_{SR}(s)$  holds for both tuples in equation (1), i.e. the subject matches the role of the task.
2.  $t1 < t2$  holds in equation (1), where ‘<’ indicates the partial order given by the PN<sup>3</sup>. SoD rules can only be enforced based on the ‘history’ of the workflow.

RB is sound if all its rules are sound.

---

<sup>3</sup> The second postulation in Definition 5 requires a partial ordering of the tasks. This only holds for a special class of PNs — loop-free nets. For PNs with loops the ‘firing history’ of the net has to be taken into account which makes the SoD rules more complex. We therefore refrain from this approach.

A sound RB of the example could contain the following rules (cf. the sample rules at the end of Section 3):

- $(smb, ay) \neg(smb, a1)$  and  $(smb, ay) \neg(smb, a2)$  plus the same rules for the two other managers Carpenter and Butcher.
- $(sma, ay) \neg(smb, a1)$  and  $(sma, ay) \neg(smb, a2)$ .
- $(fis, ay) \neg(fis, tf)$  and  $(sny, ay) \neg(sny, tf)$ .
- $(smb, a1) \neg(smb, a2)$  and  $(smb, a2) \neg(smb, a1)$  plus the same rules for the other managers.

**Definition 6 (Execution chain)** Given a start and an end marking. An *execution chain* of a PN workflow is a list of subject/task pairs whose execution (firing of the associated transitions by a legitimate subject) transforms the start marking into the end marking. If there is no contradiction to the SoD RB, the execution chain is called *SoD valid* or just *valid*.

Examples from our sample: Let the start marking be  $M(p1)=1$  (the marking of all other places 0), the end marking be  $M(p6)=1$  (the marking of all other places 0), and L1, L2, L3 be three lists with  $L1=[(fis, ay), (sma, a1), (but, a2), (sny, tf)]$ ,  $L2=[(fis, ay), (but, a1), (but, a2), (fis, tf)]$ , and  $L3=[(fis, ay), (car, a1), (but, a2), (sny, tf)]$ . L1 is no execution chain since A. Smith is no legitimate subject for one of the approve tasks (a manager is needed). L2 is an execution chain but not SoD valid since Fisher transfers his/her own travel expenses and Butcher does both approval tasks. L3 is SoD valid.

## 5 Analyzing SoD rules

In this section, we use logical reasoning to analyze the SoD rules that have been specified as part of a business process' security policy. The purpose is to find all SoD valid execution chains of a PN workflow given a sound SoD RB (cf. Definitions 3-6).

We make use of Prolog (Programming in logic) as the most popular logical programming language, to be specific GNU Prolog Version 1.1.2 by Daniel Diaz (Diaz 1999). For a comprehensive introduction to logical programming see O'Keefe (1990) and Hogger (1990).

Prolog allows for representing knowledge in a declarative, explicit, and machine independent way. A logical program is compact and flexible because it resembles more a specification than a traditional computer program. The result of a Prolog program is a logical consequence of its input (facts) using resolution with its strict mathematical background. A drawback in the use of Prolog is efficiency. Also, most implementations of Prolog mix declarative and procedural structures. Nevertheless, we will use Prolog in the first processing step due to the advantages stated above.

Figure 2 shows that our approach is divided into two processing steps. The first step uses two different input sets: the PN workflow specification and a sound SoD rule base. Subsections 5.1, 5.2 and 5.3 show how to represent these sets in Prolog in case of our example. The output of this first processing step are all valid execution chains (Section 5.4). In a second processing step, these chains are used as input for

further analysis such as workload management. Nevertheless, the focus of this paper is on processing step 1.

To clarify the following program listings, some words about Prolog notation: The number of arguments of a predicate is called *arity* and is added with a slash at the end of the predicate (e.g. `fire/3`). Names of variables start with capital letters, constants with lowercase letters. Lists are written in square brackets and we use commas to separate the individual elements (`L=[p2,p4]`). Queries start with `?-`. For more information see the GNU Prolog Manual (Diaz 1999).

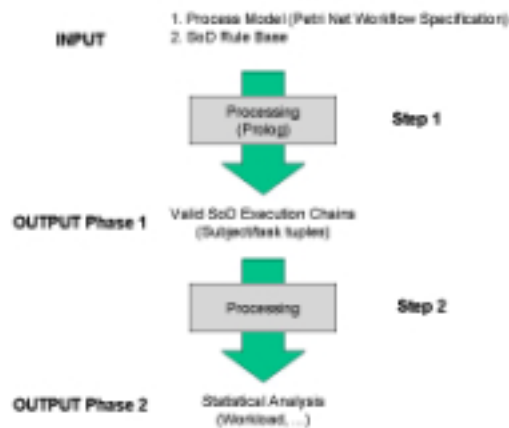


Figure 2. SoD analysis during build time

### 5.1 Petri nets in Prolog

According to Definition 1 a PN consists of places, transitions, and the flow relation. Those are represented through the predicates `transition/1`, `place/1`, and `flow/2`. Listing 1 shows the facts of our sample business case. The two predicates `preset/2` and `postset/2` give the preset and postset of a transition and return the corresponding marking (`bagof/3` is a build-in predicate that stores all possible unifications of a variable of a given expression in a list). A marking is represented as a multi set of places that in Prolog is represented as list `L`. Example: `?- postset(ay,L)` results in `L=[p2,p4]`. The `flow/2`-facts give the flow relation, i.e. define which places are connected to which transitions and vice versa. Example: `(p4,a2)` is an element of the flow relation, therefore `flow(p4,a2)` is in the listing.

The dynamic properties of a PN are expressed through the rule `fire/3`. In `fire(T,L1,L2)` it is checked if the firing of transition `T` changes marking `L1` to `L2`. This is done in several steps:

- First of all, `T` has to be a transition.
- For `T` to be activated, the preset of `T` has to be a sublist (subset) of `L1`. The `sublist/2` predicate (not shown in the listing) is used which returns true if `L` is a sublist of `L1`.

- If  $T$  fires, the marking changes: The preset of  $T$  is 'subtracted' (`setminus/3`) and the postset of  $T$  is added (`append/3`). The new marking is returned in the variable  $L3$ .

Examples: `?-fire(ay,[p1],L)` results in  $L=[p2,p4]$ , `?- fire(T,[p2,p4],[p3,p4])` results in  $T=[a1]$ , while `?-fire(ay,[p2,p4],[p6])` results in failure.

```

transition(ay). transition(a1). transition(a2). transition(tf).
place(p1). place(p2). place(p3).place(p4). place(p5). place(p6).
flow(p1,ay). flow(ay,p2). flow(ay,p4). flow(p2,a1). flow(p4,a2).
flow(a1,p3). flow(a2,p5). flow(p3,tf). flow(p5,tf). flow(tf,p6).

preset(T,L):- bagof(X,flow(X,T),L). postset(T,L):-
bagof(X,flow(T,X),L).
fire(T,L1,L2):- transition(T), preset(T,L), sublist(L,L1),
setminus(L,L1,L3),postset(T,M), append(M,L3,L4),sort0(L4,L2).

```

### Listing 1: Petri nets in Prolog

## 5.2 Organizational facts

Roles and subjects are important parts of a workflow specification. Existing roles are stored in the predicate `role/1`, subject information in the predicate `subject/1`. The predicate `play/2` allows for defining which subject can activate which role. Note that three letter abbreviations are used (man stands for the role manager, fis for the subject Fisher). Finally, roles need to be tied to tasks (transitions) which is done via the predicate `execute/2` (cf. Listing 2). Note that `play/2` and `execute/2` correspond to the functions  $f_{SR}$  and  $f_{TR}$  defined in Definition 3.

```

role(man). role(sec). role(emp).
subject(sma). subject(smb). subject(but).
subject(car). subject(fis). subject(sny).
play(sma,emp). play(smb,emp). play(but,emp). play(car,emp).
play(fis,emp). play(sny,emp). play(smb,man). play(car,man).
play(but,man). play(sny,sec). play(fis,sec).
execute(man,a1). execute(man,a2).
execute(emp,ay). execute(sec,tf).

```

### Listing 2: Organizational facts of the sample process

## 5.3 SoD data base

The predicate `separate/2` defines which task/subject tuples should be separated from others. Listing 3 shows the SoD RB for our sample process. The first six facts state that managers must not approve their own claim. The next two facts prevent B. Smith from approving the claim of his brother. Subsequently, the following two facts make sure that secretaries do not transfer the money for their own travel expenses.

Finally, the last six facts state that no manager can perform both approval tasks within the same workflow instance.

```
% Managers must not approve their own claim
separate(perform(smb,ay),perform(smb,a1)).
separate(perform(smb,ay),perform(smb,a2)).
separate(perform(car,ay),perform(car,a1)).
separate(perform(car,ay),perform(car,a2)).
separate(perform(but,ay),perform(but,a1)).
separate(perform(but,ay),perform(but,a2)).
% Brother example
separate(perform(sma,ay),perform(smb,a1)).
separate(perform(sma,ay),perform(smb,a2)).
% Secretaries must not transfer their own money
separate(perform(fis,ay),perform(fis,tf)).
separate(perform(sny,ay),perform(sny,tf)).
% Managers must not do both approves
separate(perform(smb,a1),perform(smb,a2)).
separate(perform(smb,a2),perform(smb,a1)).
separate(perform(car,a1),perform(car,a2)).
separate(perform(car,a2),perform(car,a1)).
separate(perform(but,a1),perform(but,a2)).
separate(perform(but,a2),perform(but,a1)).
```

### Listing 3: SoD database for the sample business process

## 5.4 Analysis predicates

The basis of the SoD analysis are the Prolog predicates `generate_chain/3` and `check_chain/1`. Listing 4 shows the corresponding Prolog code. The first predicate generates all possible execution chains. The definition of `generate_chain/3` uses recursion — a very powerful programming technique in Prolog. The initialization utilizes the empty list if the markings `L` and `M` are permutations of each other. In the body of the recursion it is checked whether subject `S` is able to activate the correct role and the start marking `L_s` activates transition `T`.

The predicate `check_chain/1` checks if an execution chain is SoD valid. The predicate is initialized with a single list entry `perform(S,T)` where `S` has to be a subject and `T` a transition. Then — starting with the first element — all possible separation tuples (second tuple in the separation facts) are stored in the list `L`. If this list is disjunct (checked by the predicate `disjunct/2` not shown in Listing 4) from the rest of the execution chain (`Tail`), the predicate succeeds, otherwise fails. Finally, the predicate `valid_chains/3` returns all valid SoD chains in the variable `ValidChain`.

```
generate_chain(L,M,[]):- permutation(L,M).
generate_chain(L_s,L_e,[perform(S,T)|L_v]):-
    play(S,R), execute(R,T),
    fire(T,L_s,L_t), generate_chain(L_t,L_e,L_v).
check_chain([perform(S,T)]):- subject(S), transition(T).
check_chain([H|Tail]):- bagof(X,separate(H,X),L),
    disjunct(L,Tail), check_chain(Tail).
```

```
valid_chains(StartM,EndM,ValidChain):-
    generate_chain(StartM,EndM,L),
    check_chain(L), ValidChain is L.
```

**Listing 4: Prolog predicates used for SoD analysis**

## 6 Discussion and further work

### 6.1 Discussion

The Prolog program discussed in Section 5 generates 28 possible execution chains:

```
1. [perform(sma,ay),perform(car,a1),perform(but,a2),perform(fis,tf)]
2. [perform(sma,ay),perform(car,a1),perform(but,a2),perform(sny,tf)]
...
28. [perform(sny,ay),perform(car,a1),perform(but,a2),perform(fis,tf)]
```

A first analysis yields the following:

- For the initial marking  $M(pI)=1$  and final marking  $M(p6)=1$  (all other 0) there are no deadlocks. For every employee in the example, the travel expenses workflow can be finished. No workflow has to be canceled due to too restrictive SoD rules. Although this statement may seem obvious for the example, in a more complex setting it is not.
- Every workflow requires four different persons for its execution. Without the SoD rules two persons would be sufficient (e.g. first three tasks by a manager, last one by a secretary).
- A further analysis of the valid execution chains can be used to do a ‘workload management’. In the example, the managers Carpenter and Butcher have more work because of the ‘brother rule’ (cf. Table 1).
- The analysis shows that RB of the example contains no contradicting rules.

Some words about the complexity of the analysis: The Prolog program performs a depth first search for all SoD valid execution chains of the workflow. The complexity of the program grows with the number of places/transitions of the PN, SoD rules, subjects, and roles. Consequently, the calculation for larger business processes can be very demanding concerning time and resources. We clearly understand the need for reduction techniques. Nevertheless, the analysis is designed to be done during build time of the business process, so it is not time critical. The proposed Prolog program uses the generate/test paradigm (cf. the `valid_chains/3` predicate in Listing 4). On the one hand this makes the code more comprehensible, on the other hand it is very inefficient because the generate and check predicates could be combined to cause an earlier back-tracking.

With minor modifications, the Prolog code can be used during run time, too. In most WfMSs, when a subject logs into the system, a work list is generated that shows all pending tasks for this subject. The generation of the work list can be sup-

ported by our Prolog code. With the predicates defined in Section 5, a new predicate `complete_chain/1` could be defined to complete partial execution chains. Example:

```
?-complete_chain([perform(fis,ay),perform(car,a1),X1,X2])
would result in X1=perform(but,a2), X2=perform(sny,tf) plus all
other possible completions. Note that usually a WfMS manages numerous business
processes and multiple instances of every process. Therefore an extension of the tuples
in the execution chains with an instance and process identification number is neces-
sary. Furthermore, in this case the computation would be time critical.
```

N(s,t)	sma	smb	car	but	sny	fis
ay	4	4	4	4	4	4
a1	0	8	10	10	0	0
a2	0	8	10	10	0	0
tf	0	0	0	0	14	14

**Table 1: Statistical analysis of the sample workflow.  $N(s,t)$  is the number of valid execution chains in which subject  $s$  performs task  $t$**

## 6.2 Related work

This paper introduced a SoD model based on Petri net workflows that was analyzed through a logic program. There are more complex SoD models. Ahn and Sandhu (1999) introduced RSL99 (role based separation of duties language) and Bertino et al. (1999) proposed a comprehensive language to specify and enforce authorization constraints. Going beyond our model, these languages can express constraints such as ‘a manager can only execute a specific task five times’. Furthermore, *use permissions* for operations on objects are an important issue.

Nevertheless, our rules are sufficient to ‘catch’ many of the more prominent SoD rules in today’s business processes, our model is on a comprehensible level, and — most important — the other languages are not based on Petri nets.

## 6.3 Future work

Future work will focus on the following issues:

- Knorr and Stormer (2001) introduce a tool that allows for graphically modeling SoD rules on top of an existing process definition and analyzing them as proposed in this paper. Interfaces will be provided for existing process definitions. Security officers of a company can then edit, model, simulate and analyze a security policy prior to run time of their processes.
- Our SoD model can be extended in several directions: (1) Within a task that requires access to different data items, certain privileges may be prohibited by the security policy of a company. E.g. a subject could not be allowed to change a

document that he/she created in an earlier task. (2) As indicated earlier, our SoD model depends on loop-free PNs. For general PNs, the SoD model has to take into account the history of task activations. (3) More complex SoD rules are imaginable: A subject could be eligible for a third task if he/she performed a first and not a second task. Furthermore, role hierarchies and inheritance of privileges are important features in commercial settings. A second task could only be performed by a role higher (or lower) in a hierarchy than a first task.

- An empirical study —preferably in a large company — has to show if our model is sufficient for real business processes and security policies.

## REFERENCES

- W.M.P. van der Aalst: Verification of Workflow Nets, In: Proc. of Application and Theory of Petri Nets, LNCS 1248, Springer, 1997, pp. 407-426.
- Anonymous: Internal Security, PC Week, 18(2), May 1985, pp. 89-91.
- N. R. Adam, V. Atluri and W.-K. Huang: Modeling and Analysis of Workflows Using Petri Nets, *Journal of Intelligent Information Systems* (10:2), March 1998, pp. 131-158.
- G.-J. Ahn and R. Sandhu: The RSL99 Language for Role-based Separation of Duty Constraints. In: Proc. of the Fourth ACM Workshop on Role-Based Access Control, Fairfax, VA, October 28-29, 1999.
- E. Bertino, E. Ferrari, and V. Atluri: The Specification and Enforcement of Authorization Constraints in Workflow Management Systems. *ACM Trans. on Inf. and Sys. Sec.*, 2(1):65-104, Feb. 1999.
- C. Bussler: Policy Resolution in Workflow Management Systems. *Dig. Tech. J.*, 6(4), 1995.
- A. Cichocki, A. Helal, M. Rusinkiewicz and D. Woelk: *Workflow and Process Automation — Concepts and Technology*, Kluwer Academic, 1998.
- D. Clark and D. Wilson: A Comparison of Commercial and Military Computer Security Policies. In: Proc. of the IEEE Sym. on Sec. and Privacy, pp. 184-194, Oakland, CA, 1987.
- CSI (Computer Security Institute): Issues and Trends - 1999 CSI/FBI Computer Crime and Security Survey, <http://www.gocsi.com/summary.htm>
- D. Diaz: GNU Prolog (Version 1.1.2) Manual, Edition 1.1, November 29, 1999,
- D. Georgakopoulos, M. Hornick and A. Sheth: An Overview of Workflow Management, *Distributed and Parallel Databases* (3), 1995, pp. 119-153.
- V. Gligor, S. Gavilla, and D. Ferraiolo: On the Formal Definition of Separation-of-Duty Policies and their Composition. In: Proc. of the IEEE Sym. on Sec. and Priv., 1998.
- C.J. Hogger: *Essentials of Logic Programming*, Clarendon Press, 1990.
- K. Jensen: *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use*, Volume 1, EATCS Monographs on Theoretical Computer Science, Springer, 1992.
- E. Kindler and W.M.P. van der Aalst: Liveness, Fairness, and Recurrence in Petri Nets, *Information Processing Letters* (70), 1999, pp. 269-274.
- K. Knorr and H. Stormer: Modeling and Analyzing Separation of Duties in Workflow Environments, in: Proc. of 16th IFIP/SEC, Paris, France, June 11-13 2001.
- K. Knorr: WWW Workflows Based on Petri Nets, in: Proc. of the 9th Intl. Conf. on Information Systems Development, Kristiansand, Norway, 2000.
- L. G. Lawrence: The Role of Roles, *Computers & Security*, (12) 1993, pp. 15-21.
- R. O’Keefe: *The Craft of Prolog*, MIT Press, 1990.
- C.A. Petri: *Kommunikation mit Automaten*, PhD Thesis, Universität Bonn, 1962.
- Proceedings of 5th ACM Workshop on Role-Based Access Control, Berlin, July 2000.
- W. Reisig: *Petri Nets — An Introduction*, Springer, 1985.

- R. Sandhu: Separation of Duties in Computerized Information Systems. In: Proc. of the IFIP WG 11.3 Workshop on Database Security, Halifax, UK, Sep. 1990.
- H. Stormer, K. Knorr and J. Eloff: A Model for Security in Agent-based Workflows, INFORMATIK / INFORMATIQUE, No. 6, Dec. 2000, pp. 24-29.