On-the-fly auditing of business processes

On-the-fly Auditing of Business Processes

Kees van Hee[1], Jan Hidders[2], Geert-Jan Houben[2], Jan Paredaens[3], and Philippe Thiran[4]

[1] Department of Mathematics and Computer Science
Eindhoven University of Technology
k.m.v.hee@tue.nl
[2] Department of Electrical Engineering Mathematics and Computer Science
Delft University of Technology
{a.j.h.hidders,g.j.p.m.houben}@tudelft.nl
[3] Department of Mathematics and Computer Science
University of Antwerp
jan.paredaens@ua.ac.be
[4] Faculty of Computer Science
University of Namur
philippe.thiran@fundp.ac.be

Abstract. Information systems supporting business process are mostly very complex. If we have to ensure that certain business rules are enforced in a business process, it is often easier to design a separate system, called a monitor, that collects the events of the business processes and verifies whether the rules are satisfied or not. This requires a business rule language (BRL) that allows to verify business rules over finite histories. We introduce such a BRL and show that it can express many common types of business rules. We introduce two interesting properties of BRL formulas: the future stability and the past stability. The monitor should be able to verify the business rules over the complete history, which is increasing over time. Therefore we consider *abstractions* of the history. In fact we generate from a set of business rules a labeled transition system (with countable state space) that can be executed by the monitor if each relevant event of the business process triggers a step in the labeled transition system. As long as the monitor is able to execute a step, the business rules are not violated. We show that for a sublanguage of BRL, we can transform the labeled transition system into a colored Petri net such that verification becomes independent of the history length.

1 Introduction and Motivation

In the past business information systems (BIS) were mainly used to support people in executing tasks in the business processes of an organization. Today we see that BIS have a much more responsible task: they execute large parts of the business process autonomously. So organizations become more and more dependable on their BIS. Business processes prescribe the order in which activities have to be executed. On top of that, there are often several other *business*

rules that should be met by the execution of business processes. Business rules can be required by any stakeholders, such as management, government (by law), shareholders and business partners (clients and suppliers).
It is practically impossible to verify if a BIS is satisfying the business rules. So we have to live with the fact that systems can always have some errors that may violate business rules. Normally *auditors* check periodically if the business rules are satisfied in the past period by inspecting an *audit trail* of the business process, which is in fact a log file with events that happened in the past. Not all business rule can be verified only from the past, for instance the rule: "All invoices should be paid", cannot be verified over the past in general, because there might be an unpaid invoice that but we can not conclude that it never will be paid. So *auditing* concerns only business rules that can be verified over the past. In particular we are looking for rules that have the property that they hold for an empty log and that if they become false they stay so. We call them *past stable* formulas. The example can be modified into "All invoices should be paid within 30 days" which can be audited. In this paper we take it for granted that a BIS has some unknown errors and that the system is for all stakeholders more or less a black box. Instead of a human auditor we propose here an approach where the BIS is extended by an independent *monitor system* that checks the essential business rules *on the fly* and that reports violations of business rules (detective mode) or that interrupts the BIS to prevent the occurrence of a violation (preventive mode).

We introduce a powerful language to express common business rules. Note that business rules often require summation and other hard to verify computations. To illustrate its power we express common business rules in this language. Although our *Business Rule Language* (BRL) is powerful, we have not yet developed syntactical sugar to make the language user-friendly for managers.

Our approach is to generate from a set of business rules a process model (i.e a labeled transition system) that can be executed by the monitor, in such a way that (relevant) events in the BIS are transferred to the monitor and that the monitor will try to execute the the event as a transition in the process model. As long as the monitor is able to perform the transitions, no violation of business rules should have happened and if a step is not executable this means the violation of a rule. (We do not consider here how the monitor is reset to continue after a violation.) In fact the process model of the monitor can be seen as an *abstraction* of the real business process, reflecting only the details relevant for the business rules. For on-the-fly auditing it is essential that the time to verify the rules is not increasing with the size of the log. Therefore the state space of the process model should contain all relevant information of the history to verify the business rules, therefore it is an *abstraction* of the history a the process. If the abstraction has a finite state space we are done, but in realistic situations that is not possible. The challenge is to find process models with *states* that are *bounded* in size, i.e. they should not grow with the length of the history and that have computation times dependent of the size of the states only. We first consider a standard labeled transition system, but later we transform it for a

sublangauage of BRL into a colored Petri net with bounded size of tokens and the number of tokens does not depend on the history length.

In Section 2 we give the necessary concepts for the approach and a set of characteristic business rules. In Section 3 we define BRL. In Section 4 we express some characteristic business rules in BRL. In Section 5 we first introduce two interesting properties of formulas in BRL: future stability and past stability. Then we consider the generation of a labeled transition system from the business rules. In Section 6 we show how colored Petri nets, can be used to model the labeled transition system of the monitor for sublanguages of BRL. This gives us the opportunity to use existing tools to realize the monitor.

2 Basic Concepts

Organizations perform *business processes* (also called *workflows*) in order to meet their *objectives* within the limits of a set of *business rules*. Business processes are sets of *tasks* with some precedence relationship. A business processes can have many instances concurrently. An instance of a business process is called a *case*. In a BIS many cases may be active concurrently. Tasks are executed for a case and they can be *instantaneously*, in which case the the task execution is an *event*, or the execution of a task takes time in which case we distinguish a *start event* and an *stop event*. Tasks are executed by *agents*, which can be humans or automated parts of the information system. An agent has one or more *roles* and is only allowed to execute a task if it has a role that is assigned to that task. Agents can be *authorized* by other agents to fulfil a role. We consider authorization as a special kind of task that is not case related.

The tasks manipulate objects called *resources*. Resources are created, deleted, used or transformed during a task execution. In the business process resources can be for instance materials, components, final products, energy, information or money. In a BIS the resources are always represented as data. Note that we follow the terminology that is used in *accounting information system* where the REA datamodel is used frequently. REA stands for Resource, Event and Agent, concepts with the same meaning as we gave them. The REA model is an entity relationship model for accounting information systems ([10, 9, 8]).
For the enforcement of the business rules we propose a separate *monitor system* that *detects* and reports violations of the business rules or that *prevents* violations by interrupting the BIS and triggering an exception handler. In order to do so the monitor system *records* relevant *events*, i.e. task executions of the original BIS. So the monitor is "hooked on" the original BIS and it intervenes the execution of it.

Nature of Business Rules Examples of business rules are the famous Sarbanes-Oxley [3] rules and the General Accepted Accounting Principles(GAAP). The business rules that we encounter in practice mostly are of the following nature:

- *Task-order* rules. Rules that prescribe or forbid certain task orders in a case, such as: task A should always precede task B if task C has executed.

– *Authorization* rules. Rules that prescribe or forbid certain assignments of tasks to agents or roles, such as: two specific tasks in the same case are not allowed to be executed by the same agent. This rule is known by auditors as the *four-eyes principle*. And rules that allow agents to delegate certain roles to other agents, such as an agent can only be given a certain role to another agent if he has that role himself. We call this the *delegation principle*. Other authorization rules prescribe that agents having a certain role are needed for a task.
– *Resource balancing* rules. Rules concerning the use of resources and in particular that these resources are always balanced in some form. An example is the *three-way matching rule* that prescribes that the number of items of a certain resource on an invoice should be equal to the number of items in a delivery and the number of items in the order. This rule is a well-known rule by financial auditors. Another example is that the amount of resources of a certain type is always within bounds.

Combinations of them are possible as well. Note that the task-order rules should hold for each case separately. There are case independent authorization rules as well as case dependent ones and the resource balancing rules could involve all cases.

Informal Examples of Business Rules For illustrating the business rules, we adopt a simplified version of a business process in a bakery. Here we specify the domain specific concepts. The process only the procurement of three products (i.e., resources): **butter**, **bread**, **salmon**. The procurement consists of the execution of the five following tasks: buy, packaging, packaging − grant, delivery, pay, use. The first and the last two tasks are carried out by the agent customer while the other ones are taken in charge by the agent baker. By packaging − grant, the agent baker can delegate the task packaging to his assistant (the agent assistant).

For this simple scenario, we give informal examples of each nature of business rules:

– *Task-order* rules: "for every case, the tasks buy, packaging, delivery, pay and use happen in that order"
– *Authorization* rules: "an agent with role manager can assign the role assistent to the task packaging and "for every case, the execution of the task buy is assigned to agent customer by agent baker", which in fact means that the bakery is open;
– *Resource balancing* rules: "at each moment the total amount of used bread does not exceed the amount of previously delivered bread" or "if bread is bought then it has to be delivered in the same quantity".

These examples will be used for illustrating the next sections.

3 Business Rule Language

In this section we define the language BRL to formulate business rules. Business processes are described in terms of process logs of events, which are defined

in the following. An event is characterized by the task that is performed, at a certain time, by an agent in a certain role and often for a certain case. Events can have more properties, this dependents on the task.

3.1 Postulated sets and concepts

We postulate the set $\mathbb{Q}$ of rational numbers together with the usual operations of addition ($+$), multiplication ($\cdot$) and a strict linear order ($<$). We let $X \rightharpoonup Y$ denote the set of partial functions from X to Y. For a partial function $f : X \rightharpoonup Y$ we let $\mathbf{dom}(f)$ denote the subset of X for which f is defined. The set of property types $\Theta = \{\mathbf{Time}, \mathbf{Case}, \mathbf{Task}, \mathbf{Agent}, \mathbf{Role}, \mathbf{Quantity}, \mathbf{Boolean}\}$.

Process schema A *process schema* is defined as $S = (C, T, A, R, P, \theta, \pi)$ where C is a countably infinite set of case identifiers, T is a finite set of *tasks*, A a finite set of *agents*, R a finite set of *roles* for agents, P a finite set of *properties*, $\theta : P \to \Theta$ is the *type assignment* for the properties in P and $\pi : T \to 2^P$ is the *property-set function* which gives the applicable set of properties for each task.
The set T contains at least the following tasks: $\{\text{open}, \text{close}, \text{grant}, \text{retract}\}$ where "open" means the task that starts a case and "close" the task that closes a case, and "grant" is the task of giving an agent a certain role (i.e. the right to fulfil that role) and "retract" is the inverse.
The set P contains at least the properties **time**, **case**, **task**, **agent** and **role**, such that $\theta(\mathbf{time}) = \mathbf{Time}$, $\theta(\mathbf{case}) = \mathbf{Case}$, $\theta(\mathbf{task}) = \mathbf{Task}$, $\theta(\mathbf{agent}) = \mathbf{Agent}$, $\theta(\mathbf{role}) = \mathbf{Role}$. For all $t \in T : \{\mathbf{time}, \mathbf{task}, \mathbf{agent}, \mathbf{role}\} \subseteq \pi(t)$ and for $t \in T \backslash \{\text{open}, \text{close}, \text{grant}, \text{retract}\}$ also $\mathbf{case} \in \pi(t)$. Note that the elements of the sets C and A never occur in business rules: we only use quantifications over these sets.
We distinguish a set $Res \subseteq P \backslash \{\mathbf{time}, \mathbf{case}, \mathbf{task}, \mathbf{role}, \mathbf{agent}\}$, which is used to represent *resources* and $\forall r \in Res : \theta(r) = \mathbf{Quantity}$.

The tasks with the **case** property describe the type of events that may occur in the run of a workflow, such as *place order*, *contact client*. They are domain specific and we call them *workflow tasks*. The property **case** indicates the case to which an event belongs. The property **time** denotes the timestamp of the event, i.e., when it happened. The property **task** describes the task that is performed by the event and since it defines type of the event we use the task as the name of the event. The property **agent** says which agent performed the event and **role** in which role the agent acts. In the execution of a case there can be more than one instance of a certain task. We require that each event is executed by a certain agent.

In the following definitions of this section we assume a fixed process schema $S = (C, T, A, R, P, \theta, \pi)$.

Schema-dependent concepts Based on the process schema we define several derived notions. We let $V = C \cup T \cup A \cup R \cup \mathbb{Q}$ denote the set of all possible *property values*. The semantics of the types is defined such that $[\![\mathbf{Time}]\!] = \mathbb{Q}$, $[\![\mathbf{Case}]\!] = C$, $[\![\mathbf{Task}]\!] = T$, $[\![\mathbf{Agent}]\!] = A$, $[\![\mathbf{Role}]\!] = R$, $[\![\mathbf{Quantity}]\!] = \mathbb{Q}$ and $[\![\mathbf{Boolean}]\!] = \{1, 0\} \subseteq \mathbb{Q}$.

Event We define an *event* as a partial function $ev : P \rightharpoonup V$ such that $\mathbf{task} \in \mathbf{dom}(ev)$ and if $t = ev(\mathbf{task})$ then $\mathbf{dom}(ev) = \pi(t)$ and $ev(p) \in [\![\theta(p)]\!]$ for all $p \in \pi(t)$. The set of all events is denoted as $\mathcal{E}$.

Process log A *process log* α is defined as a finite set of events. We use variables such as α and β to denote such process logs.

3.2 Core Business Rule Language

We now proceed with defining the language BRL (Business Rule Language). Given a process schema, BRL is the language in which we specify the business rules. A business rule expresses a property of the process logs.

We start with postulating a countably infinite set of variables $\mathcal{X}$ that will be used to refer to events. We then define the sets of expressions E with the following abstract syntax:

$$E ::= \neg E \mid (E \wedge E) \mid \mathcal{X}.\mathcal{P} \mid (E = E) \mid (E < E) \mid (\mathcal{X} = \mathcal{X}) \mid \mathbb{Q} \mid A \mid T \mid \Sigma(\mathcal{X} : E)\ E \mid (E + E) \mid (E \cdot E).$$

We briefly and informally describe the semantics here, and give a full formal semantics later. The expressions $\neg e_1$ and $(e_1 \wedge e_2)$ denote the logical operations over booleans. The expression $x.p$ denotes the value of property p of event x. The expression $(e_1 = e_2)$ denotes the equality operator, and $(e_1 < e_2)$ the comparison as defined by the strict linear order $<$ of $\mathbb{Q}$. The equation $(x_1 = x_2)$ is true if x_1 and x_2 refer to the same event. The numbers $q \in \mathbb{Q}$, agents $a \in A$ and tasks $t \in T$ all denote themselves. The expression $\Sigma(x : e_1)\ e_2$ denotes the summation of the value of e_2 for each event x in the process log that satisfies formula e_1. For example, $\Sigma(x : x.\mathbf{task} = \mathbf{order})\ x.\mathbf{amount}$ computes the sum of all amounts x that were ordered. Finally, the expressions $(e_1 + e_2)$ and $(e_1 \cdot e_2)$ express the addition and multiplication over $\mathbb{Q}$.

The language is subject to a typing regime, which means that we have type derivation rules that derive for some expressions a result type. If an expression has indeed such a result type then we call it *well-typed*. The expressions that are of type **Boolean** are called *formulas*. The type derivation rules are as follows. In these rules we assume that the variables $e, e_1, e_2, \ldots$ range over expressions in E, and the variable τ ranges over the types in Θ.

$$\frac{e : \mathbf{Boolean}}{\neg e : \mathbf{Boolean}} \qquad \frac{e_1 : \mathbf{Boolean} \qquad e_2 : \mathbf{Boolean}}{(e_1 \wedge e_2) : \mathbf{Boolean}} \qquad \frac{}{x.p : \theta(p)}$$

$$\frac{e_1 : \tau \qquad e_2 : \tau}{(e_1 = e_2) : \mathbf{Boolean}} \qquad \frac{\tau \in \{\mathbf{Quantity}, \mathbf{Time}\} \qquad e_1 : \tau \qquad e_2 : \tau}{(e_1 < e_2) : \mathbf{Boolean}}$$

$$\frac{}{(x_1 = x_2) : \textbf{Boolean}} \qquad \frac{q \in \mathbb{Q}}{q : \textbf{Quantity}} \qquad \frac{a \in A}{a : \textbf{Agent}} \qquad \frac{r \in R}{r : \textbf{Role}}$$

$$\frac{t \in T}{t : \textbf{Task}} \qquad \frac{e_1 : \textbf{Boolean} \quad e_2 : \textbf{Quantity}}{\Sigma(x : e_1)\ e_2 : \textbf{Quantity}} \qquad \frac{e_1 : \textbf{Quantity} \quad e_2 : \textbf{Quantity}}{(e_1 + e_2) : \textbf{Quantity}}$$

$$\frac{e_1 : \textbf{Time} \quad e_2 : \textbf{Quantity}}{(e_1 + e_2) : \textbf{Time}} \qquad \frac{e_1 : \textbf{Quantity} \quad e_2 : \textbf{Quantity}}{(e_1 \cdot e_2) : \textbf{Quantity}}$$

Note that we allow that timestamps and quantities are added, and that this results in a new timestamp. In the following of this paper we will assume that all expressions in the language are well-typed unless explicitly indicated otherwise.

We proceed with the formal definition of the semantics. A *variable binding* is defined as a partial function $\Gamma : \mathcal{X} \rightharpoonup \mathcal{E}$. For expressions $e \in E$ the semantics are defined by the proposition $\alpha, \Gamma \vdash e \rightsquigarrow v$ which states that the value of e is v for the process log α and the variable binding Γ. For a variable binding Γ and variable $x \in \mathcal{X}$ and event $ev \in \mathcal{E}$ we let $\Gamma[x \mapsto ev]$ denote the variable binding Γ' that is equal to Γ except that $\Gamma'(x) = ev$. The proposition $\alpha, \Gamma \vdash e \rightsquigarrow v$ is defined by the following rules:

$$\frac{\alpha, \Gamma \vdash e \rightsquigarrow b \qquad b \in \{0, 1\}}{\alpha, \Gamma \vdash \neg e \rightsquigarrow (1 - b)}$$

$$\frac{\alpha, \Gamma \vdash e_1 \rightsquigarrow b_1 \qquad \alpha, \Gamma \vdash e_2 \rightsquigarrow b_2 \qquad \{b_1, b_2\} \subseteq \{0, 1\}}{\alpha, \Gamma \vdash (e_1 \wedge e_2) \rightsquigarrow (b_1 \cdot b_2)} \qquad \frac{(p, v) \in \Gamma(x)}{\alpha, \Gamma \vdash x.p \rightsquigarrow v}$$

$$\frac{b \in \{0, 1\} \qquad (b = 1) \Leftrightarrow \exists_{v \in V}(\alpha, \Gamma \vdash e_1 \rightsquigarrow v \wedge \alpha, \Gamma \vdash e_2 \rightsquigarrow v)}{\alpha, \Gamma \vdash (e_1 = e_2) \rightsquigarrow b}$$

$$\frac{b \in \{0, 1\} \qquad (b = 1) \Leftrightarrow \exists_{v_1, v_2 \in \mathbb{Q}}(\alpha, \Gamma \vdash e_1 \rightsquigarrow v_1 \wedge \alpha, \Gamma \vdash e_2 \rightsquigarrow v_2 \wedge v_1 < v_2)}{\alpha, \Gamma \vdash (e_1 < e_2) \rightsquigarrow b}$$

$$\frac{b \in \{0, 1\} \qquad (b = 1) \Leftrightarrow (\Gamma(x_1) = \Gamma(x_2))}{\alpha, \Gamma \vdash (x_1 = x_2) \rightsquigarrow b} \qquad \frac{q \in \mathbb{Q}}{\alpha, \Gamma \vdash q \rightsquigarrow q} \qquad \frac{a \in A}{\alpha, \Gamma \vdash a \rightsquigarrow a}$$

$$\frac{r \in R}{\alpha, \Gamma \vdash r \rightsquigarrow r} \qquad \frac{t \in T}{\alpha, \Gamma \vdash t \rightsquigarrow t}$$

$$\frac{W = \{ev \in \alpha \mid \alpha, \Gamma[x \mapsto ev] \vdash e_1 \rightsquigarrow 1\} \quad f = \{(ev, v) \mid ev \in W \wedge \alpha, \Gamma[x \mapsto ev] \vdash e_2 \rightsquigarrow v)}{\alpha, \Gamma \vdash \Sigma(x : e_1)\ e_2 \rightsquigarrow \Sigma_{(ev,v) \in f} v}$$

$$\frac{\alpha, \Gamma \vdash e_1 \rightsquigarrow v_1 \qquad \alpha, \Gamma \vdash e_2 \rightsquigarrow v_2 \qquad \{v_1, v_2\} \subseteq \mathbb{Q}}{\alpha, \Gamma \vdash (e_1 + e_2) \rightsquigarrow (v_1 + v_2)} \qquad \frac{\alpha, \Gamma \vdash e_1 \rightsquigarrow v_1 \qquad \alpha, \Gamma \vdash e_2 \rightsquigarrow v_2 \qquad \{v_1, v_2\} \subseteq \mathbb{Q}}{\alpha, \Gamma \vdash (e_1 \cdot e_2) \rightsquigarrow (v_1 \cdot v_2)}$$

It can be observed that for each well-typed expression e such that $e : \tau$ it holds for every process log α and variable binding Γ that there is at most one v such that $\alpha, \Gamma \vdash e \rightsquigarrow v$ and if it exists then $v \in [\![\tau]\!]$. Observe that in the semantics of the summation expression $\Sigma(x : e_1)\ e_2$ the variable x is bound only to the events in α that satisfy e_1, denoted as the set W in the rule. Based on this the rule defines a partial function $f : W \rightharpoonup V$ that maps each event to the corresponding value of e_2. Then, for each element in W for which f is defined the result of f is summated. Since every process log is finite and the typing will ensure that the result of e_2 is a number if it is defined, the result of the summation is always defined.

3.3 Syntactic short-hands

We introduce syntactic short-hands for the booleans: $\mathbf{true} \equiv (1 = 1)$ and $\mathbf{false} \equiv (1 = 0)$. Moreover, we allow simultaneous quantification over several variables, i.e., we allow expressions $\Sigma(x_1, x_2, \ldots, x_n : e_1)\ e_2$ with $n > 1$ and their meaning is defined with induction on n to be equivalent with
$\Sigma(x_1 : \mathbf{true})\ (\Sigma(x_2, \ldots, x_n : e_1)\ e_2)$ for $n > 1$. So, for $n = 3$, for example, we get $\Sigma(x_1, x_2, x_3 : e_1)\ e_2 \equiv \Sigma(x_1 : \mathbf{true})\ (\Sigma(x_2 : \mathbf{true})\ (\Sigma(x_3 : e_1)\ e_2))$.

We also introduce the following short-hands for logical disjunction, logical implication, existential quantification and universal quantification: $(\varphi \vee \psi) \equiv \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \Rightarrow \psi) \equiv (\neg\varphi \vee \psi)$, $\exists x_1, \ldots, x_n(\varphi) \equiv ((\Sigma(x_1, \ldots, x_n : \varphi)\ 1) > 0)$ and $\forall x_1, \ldots, x_n(\varphi) \equiv \neg(\exists x_1, \ldots, x_n(\neg\varphi))$. For expressions that use the short-hands we generalize the notion of well-typedness such that an expression is well-typed iff the expression that is obtained after rewriting all the short-hands is well-typed. We write $a\ \theta\ b\ \theta\ c$ instead of $a\ \theta\ b \wedge b\ \theta\ c$ for $\theta \in \{<, \leq, =, >, \geq\}$.

In order to denote the first and the last event in the log we define the event expressions **last** and **first** which can be used in a formula anywhere a variable can be used. We then interpret a formula φ that contains **last** as the formula $\neg(\exists x(\mathbf{true})) \vee (\exists x((\forall y(t.\mathbf{time} \geq x.\mathbf{time})) \wedge \varphi'))$ where x and y are fresh variables and φ' is constructed from φ by replacing all occurrences of **last** with x. Analogously, if φ contains **first** it's meaning is defined as $\neg(\exists x(\mathbf{true})) \vee (\exists x((\forall y(t.\mathbf{time} \leq x.\mathbf{time})) \wedge \varphi'))$ where φ' is constructed from φ by replacing **first** with x.

4 Characteristic Examples of BRL

To illustrate BRL we present some rules from the bakery example of Section 2, but also some generic rules that fit into any business domain. Consider the following process schema $S = (C, T, A, R, P, \theta, \pi)$ with tasks $T = \{\text{open}, \text{close}, \text{grant}, \text{retract}, \text{buy}, \text{deliver}, \text{pay}, \text{use}, \text{packaging}\}$. Further C, A, R are arbitrary sets and

$$P = \{\mathbf{time}, \mathbf{case}, \mathbf{task}, \mathbf{agent}, \mathbf{role}, \mathbf{butter}, \mathbf{bread}, \mathbf{salmon}, \mathbf{to}\}$$

and the type assignment

$$\begin{aligned}\theta = \{\ & (\mathbf{time}, \mathbf{Time}), (\mathbf{case}, \mathbf{Case}), (\mathbf{task}, \mathbf{Task}), (\mathbf{agent}, \mathbf{Agent}), (\mathbf{role}, \mathbf{Role}),\\ & (\mathbf{butter}, \mathbf{Quantity}), (\mathbf{bread}, \mathbf{Quantity}), (\mathbf{salmon}, \mathbf{Quantity}),\\ & (\mathbf{to}, \mathbf{Agent})\}\end{aligned}$$

and property-set function

$$\begin{aligned}\pi = \{\ & (\text{buy}, F), (\text{deliver}, F), (\text{pay}, F), (\text{use}, F), (\text{packaging}, F),\\ & (\text{open}, F), (\text{close}, F), (\text{grant}, G), (\text{retract}, G)\}\end{aligned}$$

where the set F and G are defined as

$$\begin{aligned}G &= \{\mathbf{agent}, \mathbf{to}, \mathbf{role}, \mathbf{time}, \mathbf{task}\}\\ F &= \{\mathbf{butter}, \mathbf{bread}, \mathbf{salmon}\} \cup K\\ K &= \{\mathbf{time}, \mathbf{case}, \mathbf{task}, \mathbf{agent}, \mathbf{role}\}\end{aligned}$$

We consider the sets of characteristic examples defined in Section 2.

4.1 Task-order rules

Rule (a) The rule "no two distinct events for the same case can happen on the same moment" can be formulated as follows:

$$\forall x_1, x_2(\neg(x_1 = x_2) \Rightarrow \neg(x_1.\mathbf{time} = x_2.\mathbf{time}))$$

Rule (b) The rule "for every case the tasks buy, deliver, pay and use happen in that order." would be formulated in BRL as:

$$\begin{aligned}\forall x_1, x_2, x_3(x_1.\mathbf{task} = \text{buy} \wedge x_2.\mathbf{task} = \text{deliver} \wedge x_3.\mathbf{task} = \text{pay} \wedge\\ x_1.\mathbf{case} = x_2.\mathbf{case} = x_3.\mathbf{case}\\ \Rightarrow x_1.\mathbf{time} < x_2.\mathbf{time} < x_3.\mathbf{time})\end{aligned}$$

Rule (c) Task-order rules are very important and therefore we pay special attention to them. We define for a task a arbitrary case and a time we define:

$$\phi(a, c, t) := \Sigma(y : y.\mathbf{task} = a \wedge y.\mathbf{case} = c.\mathbf{case} \wedge y.\mathbf{time} \leq t.\mathbf{time})\ 1$$

Note that the variables represent events only. In fact a is a meta variable representing a task and C and t are representing arbitrary events, where c is used to mark an arbitrary case and t to mark an arbitrary time in the log. And we consider business rules of the form:

$$\forall c, t(\phi(a, c, t) + \phi(b, c, t) \geq \phi(d, c, t) + \phi(e, c, t))$$

where a, b, d, e are tasks. So we require for all events y, c and t that for the case of c at the time of t the number of occurrences of tasks a plus task b is at least the number of occurrences of task d and task e. Formally we can define this class of business rules by

$$\begin{aligned} E_1 &::= \phi(T, \mathcal{X}, \mathcal{X}) \mid (E_1 + E_1) \mid \mathbb{Q}. \\ E_2 &::= (E_1 \geq E_1) \mid (E_2 \vee E_2) \mid (E_2 \wedge E_2) \mid \neg E_2. \\ E_3 &::= \forall \mathcal{X}, \mathcal{X}(E_2). \end{aligned}$$

In addition we require that the two variables in E_3 must be different, appear always together and in the same order in the parameters of ϕ and that we do not allow free variables in E_3. The business rules we consider are expressions of the type E_3. Note that the formula of type E_2 should hold at each moment! This set of rules allows us to express sequences, alternatives, parallel tasks and iterations of tasks. E.g. $\forall c, t(\phi(a, c, t) \geq \phi(b, c, t))$ means that a always precedes b and $\forall c, t(\phi(a, c, t) \geq \phi(b, c, t) + \phi(d, c, t))$ that after a we have either a b or a d but not both, while $\forall x_c, x_t(\phi(a, c, t) \geq \phi(b, c, t) \wedge \geq \phi(a, c, t) \geq \phi(d, c, t))$ implies that after a, b and d may occur in parallel. The fact that multiple occurrences are allowed shows that we may have iterations. Often we require that between two occurrences of two tasks there is an occurrence of another task: $\forall c, t(\phi(a, c, t) \geq \phi(b, c, t) \geq \phi(a, c, t) - 1)$. In [12] formulas of the type E_3 are called *counting formulas* and formulas of the type $E_1 \geq E_1$ are called *basic counting formulas.*

Rule (d) A typical task order rule is using the *current time.* An example is "for all cases the task 'invoice' should be followed by a task 'payment' within 30 days". We an express this rule as:

$$\begin{aligned} \forall z(\ z.\mathbf{task} = \text{invoice} \Rightarrow \exists z'(\ & z'.\mathbf{task} = \text{payment} \wedge z'.\mathbf{case} = z.\mathbf{case} \wedge \\ & z.\mathbf{time} < z'.\mathbf{time} \leq z.\mathbf{time} + 30) \vee \\ & (z.\mathbf{time} + 30) > \mathbf{first}.\mathbf{time})) \end{aligned}$$

Note that **first** refers to the last event in the log.

Rule (e) Tasks may have a *duration* which is a property and often it is not allowed to have one agent busy in two tasks:

$$\begin{aligned} \forall x, y(\ & x \neq y \wedge x.\mathbf{agent} = y.\mathbf{agent} \Rightarrow \\ & (x.\mathbf{time} + x.\mathbf{duration} \leq y.\mathbf{time}) \vee (y.\mathbf{time} + y.\mathbf{duration} \leq x.\mathbf{time})) \end{aligned}$$

4.2 Authorization rules

Rule (f) The rule "if an agent is acting in a role then it was granted that role earlier on and that grant was not retracted in the meantime" is expressed as:

$$\begin{array}{l}\forall x(\\ \quad \exists y(\ y.\mathbf{task} = \text{grant} \wedge y.\mathbf{to} = x.\mathbf{agent} \wedge y.\mathbf{role} = x.\mathbf{role} \wedge \\ \qquad y.\mathbf{time} < x.\mathbf{time} \wedge \\ \qquad \neg \exists z(\ z.\mathbf{task} = \text{retract} \wedge z.\mathbf{to} = x.\mathbf{agent} \wedge z.\mathbf{role} = x.\mathbf{role} \wedge \\ \qquad\quad y.\mathbf{time} < z.\mathbf{time} < x.\mathbf{time})))\end{array}$$

Rule (g) The *four-eyes principle* can be formulated in general, using two arbitrary tasks T_1 and T_2:

$$\forall x, y(x.\mathbf{task} = T_1 \wedge y.\mathbf{task} = T_2 \wedge x.\mathbf{case} = y.\mathbf{case} \Rightarrow x.\mathbf{agent} \neq y.\mathbf{agent})$$

Rule (h) The *delegation principle* is formulated by two rules. The first says that an agent can grant a role to another agent only if he has the authorization himself:

$$\begin{array}{l}\forall x(\ x.\mathbf{task} = \text{grant} \Rightarrow \\ \quad \exists y(\ y.\mathbf{task} = \text{grant} \wedge y.\mathbf{to} = x.\mathbf{agent} \wedge y.\mathbf{role} = x.\mathbf{role} \wedge \\ \qquad y.\mathbf{time} < x.\mathbf{time} \wedge \\ \qquad \neg \exists z(\ z.\mathbf{task} = \text{retract} \wedge z.\mathbf{to} = x.\mathbf{agent} \wedge z.\mathbf{role} = x.\mathbf{role} \wedge \\ \qquad\quad y.\mathbf{time} < z.\mathbf{time} < x.\mathbf{time})))\end{array}$$

Rule (i) The second rule of the delegation principle says that an agent can retract a role from an agent only if he has granted this role to him before and that agent has not granted another agent:

$$\begin{array}{l}\forall x(\ x.\mathbf{task} = \text{retract} \Rightarrow \\ \quad \exists y(\ y.\mathbf{task} = \text{grant} \wedge y.\mathbf{to} = x.\mathbf{to} \wedge y.\mathbf{agent} = x.\mathbf{agent} \wedge y.\mathbf{role} = x.\mathbf{role} \wedge \\ \qquad y.\mathbf{time} < x.\mathbf{time} \wedge \\ \qquad \forall z(\ z.\mathbf{task} = \text{grant} \wedge z.\mathbf{agent} = y.\mathbf{to} \wedge z.\mathbf{role} = y.\mathbf{role} \wedge \\ \qquad\quad y.\mathbf{time} < z.\mathbf{time} < x.\mathbf{time} \Rightarrow \\ \qquad\quad \exists w(\ w.\mathbf{task} = \text{retract} \wedge w.\mathbf{agent} = z.\mathbf{agent} \wedge w.\mathbf{to} = z.\mathbf{to} \wedge \\ \qquad\qquad w.\mathbf{role} = z.\mathbf{role} \wedge z.\mathbf{time} < w.\mathbf{time} < x.\mathbf{time})))\end{array}$$

Rule (j) Often we see that an agent is claimed by a task and (hopefully) released by another task later. This means that the agent is unavailable for other tasks in the mean time. We call this an *agent reservation* rule. A special example of this is when we have pairs of events describing in fact the start and the end of a task. The tasks that mark the begin and end of a period are TaskBegin and TaskEnd. If an agent has begun a task then it cannot begin another task unless the previous task has ended.

$$\begin{array}{l}\forall x, y(\ x.\mathbf{task} = \text{TaskBegin} \wedge y.\mathbf{task} = \text{TaskBegin} \wedge x.\mathbf{agent} = y.\mathbf{agent} \wedge \\ \quad x.\mathbf{time} \leq y.\mathbf{time} \wedge x \neq y \Rightarrow \\ \quad \exists z(\ z.\mathbf{task} = \text{TaskEnd} \wedge z.\mathbf{agent} = y.\mathbf{agent} \wedge \\ \qquad x.\mathbf{time} < z.\mathbf{time} < y.\mathbf{time} \wedge x.\mathbf{case} = y.\mathbf{case} = z.\mathbf{case}))\end{array}$$

4.3 Resource balancing rules

Rule (k) In general we distinguish two kinds of resource balancing rules: *global resource balancing* rules that hold for each moment and *local resource balancing* rules that hold for each case individually and for each moment. The rule "at each moment the total amount of used bread does not exceed the amount of previously delivered bread" is a global resource balancing rule and can be formulated as:

$$\forall t(\ \Sigma(y : y.\mathbf{task} = \text{use} \wedge y.\mathbf{time} \leq t.\mathbf{time})\ y.\mathbf{bread} \leq \\ \Sigma(y : y.\mathbf{task} = \text{deliver} \wedge y.\mathbf{time} \leq t.\mathbf{time})\ y.\mathbf{bread})$$

Rule (l) The rule "for every case there is at most one delivery of bread" is a local resource balancing rule and can be formulated as:

$$\forall x, y(\ x.\mathbf{task} = \text{deliver} \wedge x.\text{bread} > 0 \wedge \\ y.\mathbf{task} = \text{deliver} \wedge y.\text{bread} > 0 \wedge x.\mathbf{case} = y.\mathbf{case} \Rightarrow x = y)$$

Rule (m) In general the resource balancing rules make use of the formulae ψ_0 for global resource balancing rules and ψ_1 for local resource balancing rules:

$$\psi_o(\mathbf{r}, t) := \Sigma(y : y.\mathbf{time} \leq t.\mathbf{time})\ y.\mathbf{r}$$
$$\psi_1(\mathbf{r}, c, t) := \Sigma(y : y.\mathbf{case} = c.\mathbf{case} \wedge y.\mathbf{time} \leq t.\mathbf{time})\ y.\mathbf{r}$$

where $\mathbf{r}$ is a resource, i.e., an element of Res. In the same way as for the task-order rules we can define a syntax:

$$E_1 ::= \psi_0(Res, \mathcal{X}) \mid \psi_1(Res, \mathcal{X}, \mathcal{X}) \mid (E_1 + E_1) \mid \mathbb{Q}.$$
$$E_2 ::= (E_1 \geq E_1) \mid (E_2 \vee E_2) \mid (E_2 \wedge E_2) \mid \neg E_2.$$
$$E_3 ::= \forall \mathcal{X}, \mathcal{X}(E_2) \mid \forall \mathcal{X}(E_2).$$

Again we also require that the two variables in E_3 are distinct and are everywhere used in the same order in the parameters of ψ.

5 Generating a Monitor from Business Rules

Our goal is to construct a monitor that gets as input each event that is processed by the system and that notifies the environment as soon as it is sure that some business rule cannot be satisfied anymore in the future. In this case the monitor will interrupt the system or it gives an alarm.

In this section we first introduce two interesting concepts, the future-stability and the past-stability. We then consider a monitor as a labeled transition system. Finally we discuss the special case where the monitor has a finite set of states. In the next section we generalize this to a Petri Net.

5.1 Stable BRL-Formula

Some BRL-formulas have a very special property, namely that once they hold then they will hold always in the future. These BRL-formula are called future-stable. So, once they hold, we need not to verify them anymore in the future. Some BRL-formula have the property that if they hold now then they held always in the past. These BRL-formula are called past-stable. So, suppose that they have to hold at an infinite number of specified moments, then we know they have to hold always. Let us define these notions exactly.

If α and β are logs, β is called a sublog of α iff $\beta \subseteq \alpha$. β is called a prelog of α, denoted $\beta \subseteq_P \alpha$ iff

- β is a sublog of α
- $\forall e_1 \in \beta\ \forall e_2 \in \beta - \alpha\ (e_1(\mathbf{time}) < e_2(\mathbf{time}))$

The closed BRL-formula f is called *future-stable* (fs) iff

$$\forall\alpha\forall\beta\ (\beta, \emptyset \vdash f \rightsquigarrow 1 \wedge \beta \subseteq_P \alpha \Rightarrow \alpha, \emptyset \vdash f \rightsquigarrow 1)$$

The closed BRL-formula f is called *past-stable* (ps) iff

$$\forall\alpha\forall\beta\ (\alpha, \emptyset \vdash f \rightsquigarrow 1 \wedge \beta \subseteq_P \alpha \Rightarrow \beta, \emptyset \vdash f \rightsquigarrow 1)$$

Example 1. Let **prop** the name of a property of type **Quantity** and a an agent.
$\exists x(x.\mathbf{agent} = a)$ is fs and $\forall x(\neg x.\mathbf{agent} = a)$ is ps;
$\forall x_1(x_1.\mathbf{time} > 0\ \Rightarrow \exists x_2(x_2.\mathbf{time} < x_1.\mathbf{time}))$ is ps;
$\forall x_1(x_1.\mathbf{time} > 0\ \Rightarrow \exists x_2, x_3(x_3.\mathbf{time} < x_2.\mathbf{time} < x_1.\mathbf{time} \wedge \neg(x_2.\mathbf{prop} = x_3.\mathbf{prop})))$ is ps;
$\forall x_1\ \exists x_2(x_1.\mathbf{time} < x_2.\mathbf{time})$ is not ps nor fs.

Corollary 1. *A BRL-formula that is both ps and fs is equivalent with 0 or 1.*

Theorem 1. *f is fs iff $\neg f$ is ps.*

Proof. Let f be fs and let $\beta \subseteq_P \alpha$ and $\alpha, \emptyset \vdash \neg f \rightsquigarrow 1$. Then $\alpha, \emptyset \vdash f \rightsquigarrow 0$, so $\beta, \emptyset \vdash f \rightsquigarrow 0$ and $\beta, \emptyset \vdash \neg f \rightsquigarrow 1$, so $\neg f$ is ps. □

Theorem 2. *If f_1 and f_2 are fs then $f_1 \vee f_2$ and $f_1 \wedge f_2$ are fs. If f_1 and f_2 are ps then $f_1 \vee f_2$ and $f_1 \wedge f_2$ are ps.*

Some BRL-formulas are not past-stable but it is still possible to find a past-stable formula that can be used by the monitor to detect when the first formula becomes definitively false. More formally, we say that formula f *is checked by* formula g if it holds that $\alpha, \emptyset \vdash g \rightsquigarrow 0$ iff $\forall_{\beta \supseteq_P \alpha}(\beta, \emptyset \vdash f \rightsquigarrow 0)$. If such a g exists then we say that f is *quasi-past-stable*. Consider for instance the two following formulas, where x_1 and x_2 are given events:

1. $\forall x_1\ \exists x_2(x_1.\mathbf{time} < x_2.\mathbf{time} \leq x_1.\mathbf{time} + 5\ \wedge\ x_1.\mathbf{prop} = x_2.\mathbf{prop})$
2. $\forall x_1\ \exists x_2(x_1.\mathbf{time} < x_2.\mathbf{time}\ \wedge\ x_1.\mathbf{prop} = x_2.\mathbf{prop})$

The first of these formulas is quasi-past-stable since it is checked by
$\forall x_1\ \forall x_2(x_1.\mathbf{time} + 5 \leq x_2.\mathbf{time} \Rightarrow \exists x_3(x_1.\mathbf{time} < x_3.\mathbf{time} \leq x_1.\mathbf{time} + 5\ \wedge\ x_1.\mathbf{prop} = x_3.\mathbf{prop}))$. The second formula is not quasi-past-stable since we always have to wait till the end of the concerned case before we can be sure that it is definitively false.

5.2 A monitor and its behaviour

Let us assume for a moment that we have a set of past-stable BRL-formulas. Their conjunction should hold. We will start with a very simple process model for the monitor: a labeled transition system that exactly obeys the BRL-formulas. So if we let the monitor system execute this process model, in the sense that each event of the BIS is the label of a transition that is executed, then we can discover violations as soon as an event occurs that brings the system in a violation state. Since the BRL-formulas are ps we know that from now on the process is violating the BRL-formulas. We model this as follows. We start with an empty process log ϵ. In fact the log so far is the *state* of the system. We assume that this initial state ϵ is a non-violation state, otherwise the whole process would violate the BRL-formulas, since they are ps. At some point in time the system is in state q. We allow a new ev event to be executed, leading to new state q'. As long as q' is a non-violation state there is no problem. But if it is a violation state, it indicates that BRL-formulas are not satisfied, and will never be satisfied in the future since they are ps.

Note that the state q of the monitor is increasing *unboundedly* and therefore the transition system can have in general an *infinite* state space Q. Since the monitor has to keep track of the BIS in real time we need a bounded aggregation of the states. In general this is not possible since we might have BRL-formulas that need to keep infinite information of the past events, such as "no two events may occur with the same value for some property p ($p \in P$)". However there are (non trivial) subsets of BRL-formulas for which bounded aggregations of the log are sufficient. We will discuss this later in this section.

We will now define the labeled transition system and the verification of a BRL-formula more formally. From now on we suppose that all events in a log have different timestamps. Consider a finite or infinite sequence of events $e_1, e_2, \ldots \in \mathcal{E}$ with $e_{i-1}(\mathbf{time}) < e_i(\mathbf{time})$ for each $i > 0$. Let $\alpha_i = \{e_1, \ldots, e_i\}$ for all $i \geq 0$. We call $M = (\mathcal{E}, Q, q_0, \delta, F)$ a *labeled transition system* where

- $\mathcal{E}$ is the set of labels which are the events.
- Q is the set of states;
- q_0 is the initial state;
- $\delta : Q \times \mathcal{E} \to Q$, the computable transition function;
- $F \subseteq Q$, the set of violation states. $Q - F$ are called the non-violation states.

We say that M *verifies* a given BRL-formula f iff there is a function $\Psi : \{\alpha_i \mid i \geq 0\} \to Q$ with

- $\Psi(\alpha_0) = q_0$;
- $\alpha_i, \emptyset \vdash f \rightsquigarrow 1 \Leftrightarrow \Psi(\alpha_i) \in F$;
- $\delta(\Psi(\alpha_i), e) = \Psi(\alpha_i \cup \{e\})$, for each event e and $i \geq 0$.

We call Ψ the *abstraction* function.

Note that $\mathcal{E}$ and Q can be infinite sets and that we have a deterministic labeled transition system. In general every BRL-formula, also those that are not

ps, can be verified by a labeled transition system. An interesting class of BRL-formulas are those that can be verified by a labeled transition system with a *finite* state space. These BRL-formulas are called *finite-state BRL-formulas*.

Each BRL-formula can be ps or not, fs or not and finite-state or not. So there are 8 possible combinations, from which one is excluded by Corollary 1. The next theorem says that the other possibilities exist:

Theorem 3. *BRL-formula that are ps and fs have to be finite-state. All the other combinations are possible.*

Proof. There are examples of the 7 possible cases: Let a be an agent, **prop** the name of a property of type **Quantity**,

1. past-stable, future-stable, finite-state: 1;
2. past-stable, future-stable, not finite-state: impossible;
3. past-stable, not future-stable, finite-state: $\forall x(\neg x.\mathbf{agent} = a)$;
4. past-stable, not future-stable, not finite-state: $\forall x_1\ \exists x_2(x_2.\mathbf{time} < x_1.\mathbf{time} \wedge x_1.\mathbf{prop} = x_2.\mathbf{prop})$
5. not past-stable, future-stable, finite-state: $\exists x(x.\mathbf{agent} = a)$
6. not past-stable, future-stable, not finite-state: $\exists x_1, x_2(x_2.\mathbf{time} = x_1.\mathbf{time} + 1 \ \wedge x_1.\mathbf{prop} = x_2.\mathbf{prop})$
7. not past-stable, not future-stable, finite-state: $\forall x_1\ \exists x_2(x_1.\mathbf{time} < x_2.\mathbf{time} \ \wedge x_1.\mathbf{agent} = x_2.\mathbf{agent})$
8. not past-stable, not future-stable, not finite-state: $\forall x_1\ \exists x_2(x_1.\mathbf{time} < x_2.\mathbf{time} \ \wedge \ x_1.\mathbf{prop} = x_2.\mathbf{prop})$

□

Theorem 4. *If f_1 and f_2 are finite-state, so are $f_1 \vee f_2$, $f_1 \wedge f_2$ and $\neg f_1$.*

Given a finite-state BRL-formula, it is easy to verify whether it is fs or ps.

Theorem 5. *Let f be a finite-state BRL-formula that is verified by the labeled transition system $M = (\mathcal{E}, Q, q_0, \delta, F)$. f is fs iff $\delta(q, e) \in F$ for every $q \in F$ and $e \in \mathcal{E}$. f is ps iff $\delta(q, e) \notin F$ for every $q \notin F$ and $e \in \mathcal{E}$.*

5.3 Incremental Verifiability

From now on we only consider past-stable BRL-formulas. Past-stable BRL-formulas have to hold permanently. So past-stable BRL-formulas f can be verified each time a new event occurs. Sometimes it is sufficient to verify f each time, sometimes it is sufficient to verify each time a simpler BRL-formula that is verifiable in constant or linear time.

Let f be a ps BRL-formula. Let the BRL-formula g be such that for each log α with prelog β and $\alpha = \beta \cup \{e\}$ it holds that $\beta, \emptyset \vdash f \rightsquigarrow 1$ implies $(\alpha, \emptyset \vdash f \rightsquigarrow 1 \Leftrightarrow \alpha, \emptyset \vdash g \rightsquigarrow 1)$. We say that f is constant verifiable iff g is computable in a constant time and that f is linear verifiable iff g is computable in time linear in the number of events in α, supposing that the events are ordered in time with the last event at the beginning of the last.

Example 2. Let **prop** the name of a property of type **Quantity** and a an agent. $\forall x(x.\mathbf{agent} = a)$ is constant verifiable, since we have only to verify whether $\mathbf{first}.\mathbf{agent} = a$. The formula $\forall x_1(x_1.\mathbf{time} > 2 \Rightarrow \exists x_2(x_2.\mathbf{time} < x_1.\mathbf{time} \wedge x_1.\mathbf{prop} = x_2.\mathbf{prop}))$ is linear verifiable, since we have only to verify that if $\mathbf{first}.\mathbf{time} > 2$ then there is an event x' with $x'.\mathbf{prop} = \mathbf{first}.\mathbf{prop}$. The formula $\forall x_1(x_1.\mathbf{time} > 2 \Rightarrow \exists x_2, x_3(x_3.\mathbf{time} < x_2.\mathbf{time} < x_1.\mathbf{time} \wedge x_2.\mathbf{prop} = x_1.\mathbf{prop} \ \wedge \ x_3.\mathbf{prop} = x_1.\mathbf{prop} + 1))$ is not constant nor linear verifiable.

Theorem 6. *If f_1 and f_2 are constant (resp. linear) verifiable, so are $f_1 \vee f_2$ and $f_1 \wedge f_2$.*

6 A Monitor based on colored Petri nets

We make in this section the the assumption that each case starts with a task "open" and ends with a task "close": a case c is open if an event x occurred with $x.\mathbf{case} = c$ and $x.\mathbf{task} = \mathit{open}$ and no event y has happened with $y.\mathbf{case} = c$ and $y.\mathbf{task} = \mathit{close}$. We assume that no event z with $z.\mathbf{case} = c$ happens before the event x or after y. If a case is not open we call it closed.

In this section we show how we can transform the set of characteristic business rules of Section 4 directly into a colored Petri net. Any labeled transition system can be expressed as a colored Petri net, so that is not remarkable. The value of these transformations is in the systematic way we derive a colored Petri net with in each state a number of tokens that is bounded by the number of "open" cases. The verification of the business rules is equivalent with firing a transition: if the transition is enabled in the net, then the rules are still valid, if not then at least one rule will be violated.

The time complexity of the computation to determine the enabling of a transition depends only on the number and size of tokens in the net and this is bounded by the number of open cases. In many practical cases we can find an upperbound for the number of open cases. So the verification of the corresponding business rules is not increasing with the size of the log, which would be the case if we would apply the brute force verification method suggested by the semantics (cf. Section 3). Note that we reduced on-the-fly auditing to simulating a colored Petri net.

6.1 Colored Petri Nets

We use here colored Petri nets for modeling purposes, so we introduce them informally. We refer to the standard literature for formal definitions: [7]. A *colored Petri net* is a 7-tuple $(P, T, F, \tau, \nu, \mu, m_0)$, where $P \cap T = \emptyset$, P is the set of *places*, T the set of *transitions*, $F \subseteq (P \times T) \cup (T \times P)$ the set of *arcs*, τ is a function with $\mathrm{dom}(\tau) = P$ and for $p \in P : \tau(p)$ is a type called *color set*, ν is a function with $\mathrm{dom}(\nu) = F$ and for each $f \in F : \nu(f)$ is an expression, called *arc inscription*, μ is a function with $\mathrm{dom}(\mu) = T$ and for each $t \in T : \mu(t)$ is a predicate called a *guard* and finally m_0 is the *initial state* (initial marking).

Note that if we discard the functions τ, ν and μ then we have a classical Petri net. In CPN tools (cf. [7]) there is a syntax for color sets, arc inscriptions and guards. We deviate a little from these official syntax, because we have only very simple expressions and we prefer to use standard mathematical notations. The arc inscriptions we consider are variables from the set $V = C \cup T \cup A \cup R \cup \mathbb{Q}$ defined in Section 3, simple *arithmetical* expressions with these variables and constants of $\mathbb{Q}$ and *vectors* of these expressions (e.g. $(a, r, n + 1)$ for $a \in A \wedge \ r \in R \wedge \ n \in \mathbb{Q}$. The guard of a transition is determined by pattern matching between the inscriptions of the arcs connected to the transition and some explicit predicates. The enabling rule is as usual: if for each input arc we can select a token with such values that we can find a binding of the free variables making the guard becomes true, then the transition can fire, the input tokens are consumed and for each output arc a token is produced by the binding of the variables in the arc inscriptions. In Fig. 1 we see a transition F with input places A with type $\mathbb{Q}$ and B with type $\mathbb{Q} \times \mathbb{Q}$ and output places C with type $\mathbb{Q}$ and D with type $\mathbb{Q} \times \mathbb{Q} \times \mathbb{Q}$. The guard says $x \neq y \wedge \ z = 2.x + y$ which means that F is not enabled if $x = y$ and if it is enabled it may fire. If it fires it consumes tokens from places A and B with values say a and (a, b) respectively $(a \neq b)$ and it produces output tokens for C and D with values $b = 2$ and $(a, b, 2.a + b)$ respectively.

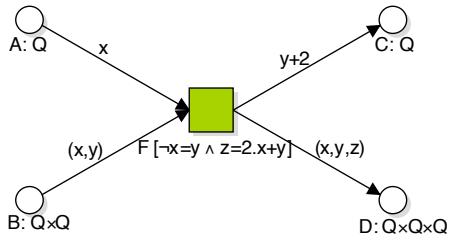

Fig. 1. Transition with arc inscriptions and guard

We also use the concept of a *workflow net*, a special class of Petri nets (cf. for details see [2, 1]). Here we consider a Petri net to be a workflow net if there is exactly one transition without input arcs (called "open"), exactly one transition without output arcs (called "close") and every other place or transition is on a directed path from "open" to "close". We consider colored Petri nets in which subnets have the structure of a workflow net.
It is well-known (cf. [7]) to define a labeled transition system for a colored Petri net.

In the following sections we define constructions in the form of colored Petri nets for the five kinds of characteristic business rules: task-order rules, resource balancing rules, delegation principle, the four eyes principle and the agent reser-

vation rule. We consider these constructions as *patterns*. They have to be applied one after the other. We introduce these constructions in an informal way.

6.2 Task-order rules

Here we only consider *counting formula* (cf. Section 4). In [12] it is shown that *each* counting formula can be represented in a *disjunctive normal* form $\forall c, t\ (\bigvee_i \bigwedge_j \theta_{i,j})$ where $\theta_{i,j}$ is a *positive basic counting* formula, i.e. without negations. In [11] it is shown how counting formula can be transformed into classical Petri nets with inhibitor arcs and multiple transitions with the same label. Although the same constructs can be applied here, we restrict us here to a subset of business rules that are counting formula of the form: $\forall c, t\ (\bigwedge_i \theta_i)$ where θ_i is a positive *basic counting* formula without constants and only $\geq$ signs. The main reason for this restriction is that we have only *past-stable formulas*. As an example consider:

$$\forall c, t((\phi(a, c, t) + \phi(b, c, t) \geq \phi(d, c, t) + \phi(e, c, t) \wedge \phi(b, c, t) \geq \phi(g, c, t))$$

where a, b, d, e, g are tasks. We will reduce such a business rule by eliminating redundancies. Reduction means that if we have a subformula θ in the conjunction that is implied by one or more other subformula, then we delete θ. For example if we have $\forall c, t((\phi(a, c, t) \geq \phi(b, c, t) \wedge \phi(b, c, t) \geq \phi(d, c, t) \wedge \phi(a, c, t) \geq \phi(d, c, t))$ then we delete $\phi(a, c, t) \geq \phi(d, c, t)$. Another example of reduction is $\forall c, t((\phi(a, c, t) + \phi(b, c, t) \geq \phi(d, c, t) \wedge \phi(b, c, t) \geq \phi(d, c, t) + \phi(e, c, t))$ then we delete $\phi(a, c, t) + \phi(b, c, t) \geq \phi(d, c, t)$. So the reduction rule keeps the formula with the *minimal* number of terms on the left hand side of the $\geq$ sign and the *maximal* number of terms on the righthand side of the $\geq$ sign. We assume that the task-order rules are in the normal form described above and are irreducible. The construction rule says that we create for each basic counting formula a place that is an *output* place of all transitions on the left hand side and an *input* place for all transitions on the righthand side of the $\geq$ sign. In Fig. 2 we see the transformation of $\forall c, t(\phi(a, c, t) + \phi(b, c, t) \geq \phi(d, c, t) + \phi(e, c, t))$ into the Petri net with one place called p and four transitions $\{a, b, d, e\}$.

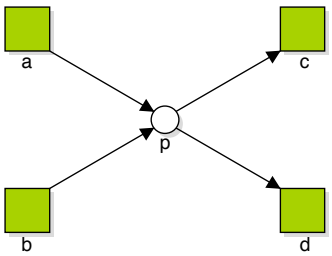

Fig. 2. Transformation of a basic counting formula

If we have a business rule that is a conjunction of a set of basic counting formulas then we repeat this construction and so we obtain a (classical) Petri net. Note that since we consider only the conjunction of business rules we may combine the task-order rules into one business rule. We mention some properties that that can be verified by standard Petri net methods (cf.[5]).

- The Petri net obtained by the task-order rules is a workflow net.
- For each reachable marking after the "open" transition has fired once, there exists (not necessarily in the log) a firing sequence that fires transition "close" once and then after that no transition can fire, so it is a *deadlock*.
- In addition to the former property, there are no tokens left in the net. This property is called proper termination or *soundness* (cf. [1]).

If the Petri net satisfies the third assumption then we know that after firing the "close" transition of a case, we know that the verification of the enabling of transitions involves only the open cases. Next we will augment our net with color:

- All places in this net obtain the case identities, i.e. C, as color set.
- Transition "open" generates with each firing a new case identity and this case identity is the value of all the tokens produced by "open". (This can be realized by standard methods).
- All transitions have as inscriptions on their input and output arcs the same case variable (from C) which means that transitions can only fire if they consume tokens with the same case identity and that they produce tokens with the same case identity.

So we obtain a colored Petri net of which Fig. 3 is an example. The formula that is represented by the net is:

$$\forall c, t(\phi(open, c, t) \geq \phi(a, c, t) + \phi(g, c, t) \wedge \phi(a, c, t) + \phi(f, c, t) \geq \phi(b, c, t) \wedge$$

$$\phi(b, c, t) \geq \phi(d, c, t) \wedge \phi(d, c, t) \geq \phi(e, c, t) + \phi(f, c, t) \wedge$$

$$\phi(e, c, t) + \phi(q, c, t) \geq \phi(close, c, t) \wedge$$

$$\phi(g, c, t) \geq \phi(h, c, t) \wedge \phi(h, c, t) \geq \phi(j, c, t) \wedge \phi(h, c, t) \geq \phi(k, c, t) \wedge$$

$$\phi(j, c, t) \geq \phi(l, c, t) \wedge \phi(k, c, t) \geq \phi(m, c, t) \wedge \phi(l, c, t) \geq \phi(n, c, t) \wedge$$

$$\phi(m, c, t) \geq \phi(n, c, t) \wedge \phi(n, c, t) \geq \phi(q, c, t))$$

Here we have in fact two workflows, the upper part of the figure and the lower part represent both workflows. So is it possible to combine arbitrary many workflows into one workflow net. The task-order rules are valid if and only if the firing sequence from the event log is executable on the Petri net. Note that in the language-based based theory of regions and in process discovery the same idea of transforming a set of equations over firing sequences into a place is used (cf. [4, 13]).

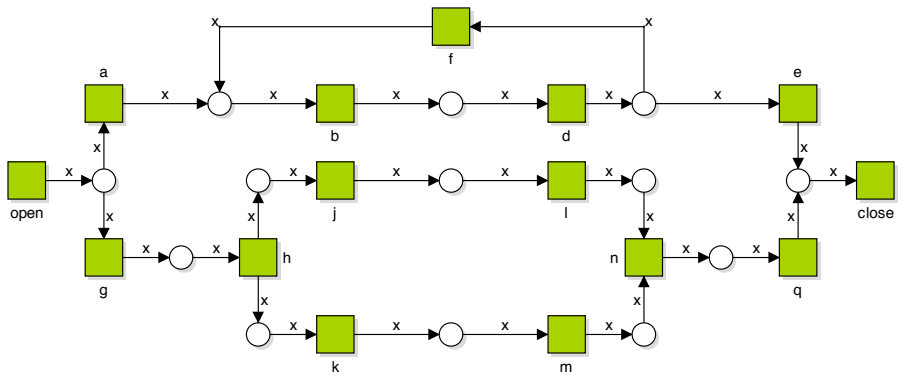

Fig. 3. Example of a workflow net with two case types

6.3 Resource balancing rules

The next step is to add the resource balancing rules. We have two kinds of resource balancing rules, global and local resource balancing rules. We consider two examples: $\forall t(l \leq \psi_0(res_1, t) + \psi_0(res_2, t) \leq u)$ for global resources res_1 and res_2 and $\forall c, t(l \leq \psi_1(res_3, c, t) \leq u)$, where l and u are constants representing upper and lower bounds.

For each global resource property concerning resource res we simply add (outside of the workflow) one place with label res and type $\mathbb{Q}$ that has one token in the initial state. Any task t having a global resource as property will have this resource place as input as well as output place. The inscription on the input arc is a variable, say x, from $\mathbb{Q}$ and the inscription of the output arc is $x + y$ if an event occurs for task t with for property res the value y.

To model the coupling of the monitor to the BIS we add to the workflow net, augmented with resource places, a special transition called "event handler" with one output place that stores event data in a vector format. It filters out properties as **time**, but the **task** and **case** and resources are there in a fixed order. In the left diagram of Fig. 4 we see two global resources $res1$ and $res2$ and their connections to the tasks. We have not shown in the diagram that both transitions have an additional guard $l \leq x + y + r_1 + r_2 \leq u$ to express the example rule given above. Transition $task1$ adds the value of r_1 to the token in place $res1$ and transition $task2$ adds in addition the value r_2 to the token in place $res2$. We also see the event handler with one output place that is input place for all transitions in the workflow. Each transition in the workflow has a guard that requires that an event token, i.e., the token generated by the event handler, needs to have a task attribute that is the same as the label of the transition. So the event data can be bound to the arc inscriptions of the transition that represents the task.

In Fig. 4 we see in the right diagram a local resource. Local resources are bound to cases. This can be expressed by connecting the resource places to

the "open" and "close" transitions: "open" puts the resource token in the local resource place and "close" deletes it. Further the transitions are connected as global resource places. The additional guard for transition $task3$ is $l \leq x+r_3 \leq u$.

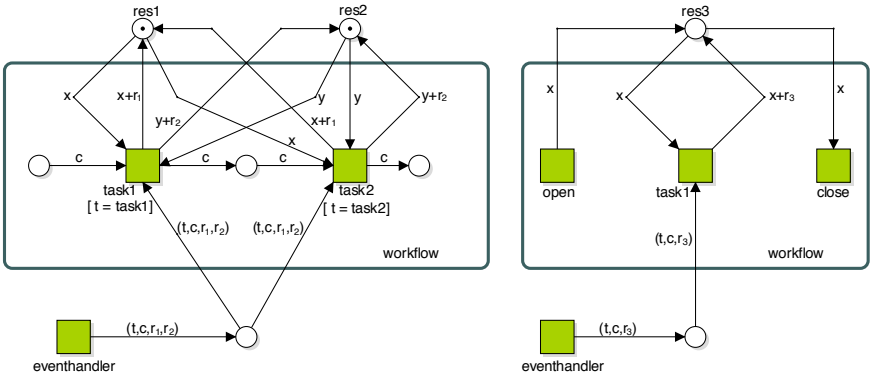

Fig. 4. Event handler, global and local resources

It is easy to see that if the resource balancing rules hold in some state of the Petri net, then they continue to hold after a transition (if it can fire). Note that the event handler and its output place will not be implemented in the monitor: they are here to show how the connection with the BIS can be modeled.

6.4 The delegation principle

In Fig. 5 we see a subnet that is outside of the workflow with two tasks "grant" and "retract" and two places "agent-roles" and "agent-delegate". The arc inscriptions explain the working: if "grant" fires it adds to place "agent-roles" a token with value (a, b, r) meaning that agent a receives from agent b the role r. In order to be able to do so it needs a token with value (b, c, r) indicating that agent b has been granted by some agent c for role r. Simultaneous a token with value (b, r, n) is consumed from place "agent-delegate" meaning that agent b has granted role r to other agents already n times. Also a new token is produced for this place with the update, namely $(b, r, n + 1)$. The working of task "retract" a similar. In the initial state we need at least for each role one token in place "agent-roles" with an agent that has that role (for example a role "manager" that has all roles). In place "agent-delegate" we have in the initial state as many tokens as there are agents. It is easy to see that agents only are granted a role by somebody who has that role, and that roles are only retracted from agents that have no outstanding grants of that role for other agents. This subnet should be connected to each transition in the workflow net representing a task that needs an agent in a role: one arc as input and one as output for that task. This expresses that the agent in that role is needed for the task. Note that we assume

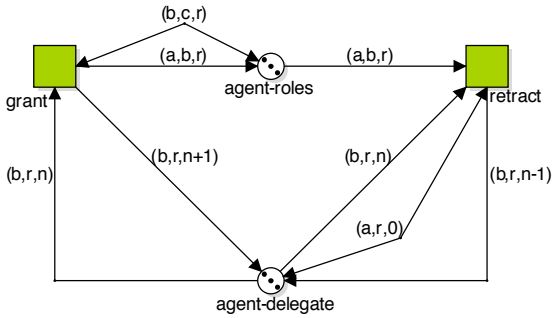

Fig. 5. Delegation principle

that an agent can have more than one role at the same time. Also the transitions "grant" and "retract" are thought to be connected to the 'eventhandler as we have seen above for instance to determine that agent b is granting agent a.

6.5 Four eyes principle and agent reservation rule

Next we show how the four eyes principle can be represented. In the left diagram of Fig. 6 we see two tasks of the workflow ("task1" and "task2"). We assume that "task1" has to come before "task2", although it is not required that "task2" is an immediate successor of "task1". We also assume that both tasks are executed both only once. For more general situations the subnet becomes more complicated but the idea is the same. We added, outside of the workflow, a place "agents" where for each agent there is one token. This place is connected with input and output arcs with the tasks. Place "p" is inside the workflow and contains a token with value (c, a) where c is the case identity and a the agent that has been performing "task1". The guard for "task2" guarantees that in "task2" not the same agent is involved as in "task1" for this case.
In the right diagram of Fig. 6 we see a similar construct for agent reservation. Here it is assumed that in "task1" an agent a is needed and that this agent remains involved until he is released in "task2". The difference with the four eyes pattern is that the connections with place "agents" are single: "task1" consumes a token from this place and "task2" produces one for it. It is easy to see that if the rules are valid and a transition can fire they remain valid.

Note that all these patterns can be combined in one colored Petri net. For the last two sets of rules the number of tokens in the subnet is fixed, by the number of agents and the number of roles. In task-order rules the number of tokens depends only on the number of open cases. For global resources we have only one token per resource place and for the local resources we have In the resource balancing rules one token per resource per case. So the total number of tokens in the net is independent of the length of the history, which make the approach suitable for on the fly auditing.

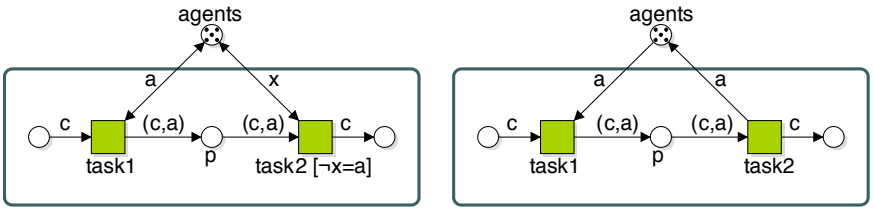

Fig. 6. Four eyes principle and agent reservation

7 Conclusion and future work

We have presented a language for business rules (BRL) and an approach to build a monitor system that is able to check business rules on the fly in parallel to a business information system (BIS). The assumption is that we can not trust the BIS. We have seen how we evaluate subsets of business rules by executing a colored Petri net, and that the computations involved are not depending on the length of the event log. If the Petri net can not execute a transition, then a rule is violated and this can be reported or it may generate an interrupt for the BIS. As future work we will try to find more classes of business rules that can be translated to Petri net patterns and we are looking for a more systematic approach. We are involved in a case study in practice using an implementation based on CPN tools (cf. [6]).

References

1. W.M.P. van der Aalst, *Verification of workflow nets*, ICATPN 1997 (London, UK), Springer-Verlag, 1997, pp. 407–426.
2. W.M.P. van der Aalst and K.M. van Hee, *Workflow Management: Models, Methods and Systems (in Dutch)*, Academic Service, Schoonhoven, 1997.
3. D. Berg, *Turning sarbanes-oxley projects into strategic business processes*, Sarbanes-Oxley Compliance Journal (2004).
4. Ph. Darondeau, *Deriving Unbounded Petri Nets from Formal Languages*, CONCUR 1998 (London, UK), Springer-Verlag, 1998, pp. 533–548.
5. Girault, C. and Valk, R., *Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications*, Springer.
6. K. Jensen and L.M. Kristensen, *Coloured Petri nets : modelling and validation of concurrent systems*, Springer, 2009.
7. Kurt Jensen, *Coloured petri nets: basic concepts, analysis methods and practical use, vol. 2*, Springer-Verlag, London, UK, 1995.
8. W.E. McCarthy, *The rea accounting model: a generalized framework for accounting systems in a shared dat environment*, The accounting review (1982).
9. _____, *An ontology analysis of the primitives of extended rea enterprise information architecture*, International Journal of Accounting Information Systems **3** (2002).

10. P.J. Romney, M.B.;Steinbart, *Accounting information systems*, 11 ed., Pearson International Editions, 2009.
11. K. M. van Hee, N. Sidorova A. Serebrenik, and W. van der Aalst, *Working with the past: integrating history in petri nets*, Fundamenta Informaticae (2004).
12. K. M. van Hee, A. Serebrenik, N. Sidorova, and W. M. P. van der Aalst, *History-dependent Petri nets*, Petri Nets and Other Models of Concurrency - ICATPN 2007 (Jetty Kleijn and Alex Yakovlev, eds.), Springer, 2007.
13. Jan Martijn E.M. van der Werf, Boudewijn F. van Dongen, Cor A.J. Hurkens, and Alexander Serebrenik, *Process Discovery using Integer Linear Programming*, ATPN (van Hee, Kees M. and Rüdiger Valk, eds.), LNCS, vol. 5062, Springer, 2008, pp. 368 – 387.