

On-the-fly Auditing of Business Processes

Kees van Hee¹, Jan Hidders², Geert-Jan Houben², Jan Paredaens³, and
Philippe Thiran⁴

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 processes are usually 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. Actually 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 whether 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 whether the business rules are satisfied in the past period by inspecting an *audit trail* of the business process, which is actually a log file with past events. Not all business rules 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 for which we cannot conclude that it will never 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 are false for the current log then they stay false for every future extension of that log. 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 assume that a BIS may have some unknown errors and that the system is for all stakeholders 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 may consider the BIS extended with the monitor system as a new BIS. This BIS has a better chance of preserving the business rules than the original one. The main reason is that the monitor system is an independent subsystem dedicated to the verification of the business rules.

We introduce a powerful language to express common business rules. Note that business rules often require summation and other computations. To illustrate its power we express common business rules in this language.

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 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 of 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 size of the log 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 sublanguage 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 concurrent instances. 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. They can be *instantaneous*, in which case 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 fulfill a role. We consider authorization as a special kind of task that is not case related.

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 the domain of *accounting information systems* 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 datamodel is an Entity Relationship model for accounting information systems [1–3].

For the enforcement of the business rules we propose a *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 [4] rules and the International Accounting Standards [5]. 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 been 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*. Other examples are 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 these resources that are balancing in some way. An example is the *three-way matching rule* that prescribes that the number of items of a certain resource in an invoice should be equal to the number of items in a delivery and the number of items in the order. This rule is well-known by financial auditors. Another example is that the amount of resources of a certain type has always to be 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 describes the procurement of three products, i.e., resources: **butter, bread, salmon**. The procurement consists of the execution of the six following tasks: buy, packaging, packaging – grant, delivery, pay and 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 assistant to the task packaging” and “for every case, the execution of the task buy is assigned to agent customer by agent baker”, which actually 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 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 depends 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 $X \rightarrow Y$ denote the set of partial functions from X to Y . For a partial function $f : X \rightarrow Y$ we let $\mathbf{dom}(f)$ denote the subset of X for which f is defined. The property types set is $\Theta = \{\mathbf{Time}, \mathbf{Case}, \mathbf{Task}, \mathbf{TaskType}, \mathbf{Agent}, \mathbf{Role}, \mathbf{Quantity}, \mathbf{Boolean}\}$.

Process schema. A *process schema* is defined as $S = (C, T, A, R, \mathcal{P}, \theta, \pi)$ 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, \mathcal{P} a finite set of *properties*, $\theta : \mathcal{P} \rightarrow \Theta$ is the *type assignment* for the properties in \mathcal{P} and $\pi : T \rightarrow 2^{\mathcal{P}}$ is the *property-set function* which gives the applicable set of properties for each task.

The set \mathcal{P} contains at least the properties **time**, **case**, **task**, **tasktype**, **agent**, **to** and **role**, such that $\theta(\mathbf{time}) = \mathbf{Time}$, $\theta(\mathbf{case}) = \mathbf{Case}$, $\theta(\mathbf{task}) = \mathbf{Task}$, $\theta(\mathbf{tasktype}) = \mathbf{TaskType}$, $\theta(\mathbf{agent}) = \mathbf{Agent}$, $\theta(\mathbf{to}) = \mathbf{Agent}$ and $\theta(\mathbf{role}) = \mathbf{Role}$.

The set $\llbracket \mathbf{TaskType} \rrbracket = \{\text{open, close, grant, retract, engage, release, use}\}$ is the set of task types. The task-types have the following meaning: “open” means that the task starts a case and “close” that the task closes a case, “grant” is a task of giving an agent a certain role, i.e., the right to fulfill that role, and “retract” is the inverse. Further “engage” means that an agent is engaged for a case and “release” that it is freed. A task of type “use” is only allowed to use an agent that is engaged for the case. In this way we can model that activities take time: so an activity may start with an engage task, then a use task is executed and it ends with a release task. The time between the engage task and the release task is the time the agent is busy. With “engage” and “release” we can also model more complex structures.

For all $t \in T : \{\mathbf{time}, \mathbf{task}, \mathbf{tasktype}, \mathbf{agent}, \mathbf{role}\} \subseteq \pi(t)$. We sometimes have “dummy” tasks, called “dummy” for instance when we open or close a case. Such tasks are characterized by their task type and they have no additional properties. 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 $Res \subseteq \mathcal{P} \setminus \{\mathbf{time}, \mathbf{case}, \mathbf{task}, \mathbf{tasktype}, \mathbf{role}, \mathbf{agent}, \mathbf{to}\}$, 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 *case 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. Finally property **to** says to which agent a role is delegated. 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 $S = (C, T, A, R, \mathcal{P}, \theta, \pi)$.

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 $\llbracket \mathbf{Time} \rrbracket = \mathbb{Q}$, $\llbracket \mathbf{Case} \rrbracket = C$, $\llbracket \mathbf{Task} \rrbracket = T$, $\llbracket \mathbf{Agent} \rrbracket = A$, $\llbracket \mathbf{Role} \rrbracket = R$, $\llbracket \mathbf{Quantity} \rrbracket = \mathbb{Q}$ and $\llbracket \mathbf{Boolean} \rrbracket = \{1, 0\} \subseteq \mathbb{Q}$. Note that $\llbracket \mathbf{TaskType} \rrbracket$ is already defined.

Event. We define an *event* as a partial function $ev : \mathcal{P} \rightarrow V$ that maps properties to their value such that (i) at least the property **task** which indicates the executed task has a value, i.e., $\mathbf{task} \in \mathbf{dom}(ev)$, (ii) exactly all properties that are associated with the task of the event have a value, i.e., if $t = ev(\mathbf{task})$ then $\mathbf{dom}(ev) = \pi(t)$ and (iii) the value of each property is in the semantics of the type of that property, i.e., $ev(p) \in \llbracket \theta(p) \rrbracket$ for all $p \in \pi(t)$. The set of all events is denoted as \mathcal{E} . So an $ev \in \mathcal{E}$ can be seen as a record of an event that occurred in the BIS.

Event log. A *event log* α is defined as a finite set of events. We use variables such as α and β to denote such event logs. Note that although they are sets, event logs can be thought of as ordered sets where the events are ordered by their timestamp.

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 event logs.

We start by 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 in Appendix A. 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} which is used to represent both quantities and time. 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 event log that satisfies formula e_1 . For example, $\Sigma(x : x.\mathbf{task} = \mathit{order}) x.\mathbf{amount}$ 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} . In order to avoid expressions that have no well-defined result, such as boolean operations over non-boolean values, there is a typing regime that defines when an expression is *well-typed*, which is formally defined in Appendix A.

3.3 Syntactic short-hands

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

$$\Sigma(x_1 : \mathbf{true}) (\Sigma(x_2, \dots, x_n : e_1) e_2)$$

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: $(\varphi \vee \psi) \equiv \neg(\neg\varphi \wedge \neg\psi)$, $(\varphi \Rightarrow \psi) \equiv (\neg\varphi \vee \psi)$, $\exists x_1, \dots, x_n(\varphi) \equiv ((\Sigma(x_1, \dots, x_n : \varphi) 1) > 0)$ and $\forall x_1, \dots, x_n(\varphi) \equiv \neg(\exists x_1, \dots, x_n(\neg\varphi))$. 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 let $e_1 \leq e_2$ and $e_1 \geq e_2$ denote $\neg(e_1 > e_2)$ and $\neg(e_1 < e_2)$, respectively. We write $e_1 \mathit{op}_1 e_2 \mathit{op}_2 e_3$ instead of $(e_1 \mathit{op}_1 e_2) \wedge (e_2 \mathit{op}_2 e_3)$ for $\mathit{op}_1, \mathit{op}_2 \in \{<, \leq, =, >, \geq\}$. Finally we introduce a short-hand to denote an enumeration of possible values. We let $x.p \in \{v_1, \dots, v_n\}$ denote the formula $x.p = v_1 \vee \dots \vee x.p = v_n$.

In order to denote the first and the last event in the log, assuming the ordering of events by their timestamp, we define the event expressions **first** and **last** which can be used in a formula anywhere a variable can be used. We then interpret a formula φ that contains **first** as the formula $\neg(\exists x(\mathbf{true})) \vee \exists x(\forall y(y.\mathbf{time} \geq x.\mathbf{time}) \wedge \varphi')$ where x and y are fresh variables and φ' is constructed from φ by replacing all occurrences of **first** with x . Analogously, if φ contains **last** its meaning is defined as $\neg(\exists x(\mathbf{true})) \vee \exists x(\forall y(y.\mathbf{time} \leq x.\mathbf{time}) \wedge \varphi')$ where φ' is constructed from φ by replacing **last** 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 $S = (C, T, A, R, \mathcal{P}, \theta, \pi)$ with tasks

$$T = \{\text{buy, deliver, pay, use, packaging, dummy}\}$$

Further C, A, R are arbitrary sets and

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

The last three are resources. The type assignment is:

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

and the property-set function

$$\pi = \{(\text{buy}, F), (\text{deliver}, F), (\text{pay}, F), (\text{use}, F), (\text{packaging}, F), (\text{dummy}, G \cup \{to\})\}$$

where the set F is defined as $F = \{\mathbf{case, butter, bread, salmon}\} \cup G$ and $G = \{\mathbf{agent, role, time, task, tasktype}\}$.

We consider the sets of characteristic examples defined in Section 2. Note that all rules should hold at every moment, so also for every prefix of an event log.

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:

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

Rule (c). Task-order rules are very important and therefore we pay special attention to them. We first define a short-hand for counting the number of occurrences of task a in the case of event c :

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

Note that the variables c and y represent events only. Actually a is a meta variable representing a task and c is representing an arbitrary event used to mark an arbitrary case in the log. Now, consider for example business rules of the form:

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

where a, b, d, e are tasks. These require that for all cases the number of occurrences of tasks a and b is at least the number of occurrences of task d and task e . Because we cannot quantify over case identifiers, we quantify over all events c and we count the number of events y with the same case identifier as c and task equal to a, b, d and e respectively. So we consider c as a dummy variable over the case identifiers. Formally we define a class of similar business rules by

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

In addition we do not allow free variables in E_3 . Business rules of type E_3 can express sequences, alternatives, parallel tasks and iterations of tasks. For example $\forall c(\phi(a, c) \geq \phi(b, c))$ means that a always precedes b and $\forall c(\phi(a, c) \geq \phi(b, c) + \phi(d, c))$ that after a single a we may have either a b or a d but not both, while $\forall c(\phi(a, c) \geq \phi(b, c) \wedge \phi(a, c) \geq \phi(d, c))$ implies that after a single a , b and d may occur both together. 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: $\forall c(\phi(a, c) \geq \phi(b, c) \geq \phi(a, c) - 1)$. In [6] 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 uses the *current time*. An example is “for all cases the task ‘invoice’ should be followed by a task ‘payment’ within 30 days”. We can express this rule as:

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

Recall that **last** refers to the last event in the log.

Rule (e). Cases have a start and a stop event, which is expressed by two task types called “open” and “close”. No event for a case is allowed to happen before the start event or after the stop event.

$$\begin{aligned} \forall x(\neg(x.\mathbf{tasktype} \in \{\mathbf{open}, \mathbf{close}, \mathbf{grant}, \mathbf{retract}\}) \Rightarrow \\ ((\exists y(y.\mathbf{tasktype} = \mathbf{open} \wedge y.\mathbf{case} = x.\mathbf{case} \wedge y.\mathbf{time} \leq x.\mathbf{time})) \end{aligned}$$

\wedge

$$\begin{aligned} \forall x(\neg x.\mathbf{tasktype} \in \{\mathbf{open}, \mathbf{close}, \mathbf{grant}, \mathbf{retract}\}) \wedge \\ \exists y(y.\mathbf{tasktype} = \mathbf{close} \wedge y.\mathbf{case} = x.\mathbf{case}) \Rightarrow x.\mathbf{time} \leq y.\mathbf{time}) \end{aligned}$$

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:

$$\forall x(x.\mathbf{time} > 0 \wedge \exists y(y.\mathbf{tasktype} = \mathbf{grant} \wedge y.\mathbf{to} = x.\mathbf{agent} \wedge \\ y.\mathbf{role} = x.\mathbf{role} \wedge y.\mathbf{time} < x.\mathbf{time} \wedge \\ \neg \exists z(z.\mathbf{tasktype} = \mathbf{retract} \wedge z.\mathbf{to} = x.\mathbf{agent} \wedge \\ z.\mathbf{role} = x.\mathbf{role} \wedge y.\mathbf{time} < z.\mathbf{time} < x.\mathbf{time})))$$

We assume at time=0 several initial events that grant roles to agents. The agent of these events could be the manager. We do not give these events here.

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

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

Note that this is just an example of the four-eyes principle. We may have rules that require that all tasks in some set of tasks should have different agents per case.

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:

$$\forall x(x.\mathbf{tasktype} = \mathbf{grant} \Rightarrow \\ \exists y(y.\mathbf{tasktype} = \mathbf{grant} \wedge y.\mathbf{to} = x.\mathbf{agent} \wedge y.\mathbf{role} = x.\mathbf{role} \wedge \\ y.\mathbf{time} < x.\mathbf{time} \wedge \\ \neg \exists z(z.\mathbf{tasktype} = \mathbf{retract} \wedge z.\mathbf{to} = x.\mathbf{agent} \wedge z.\mathbf{role} = x.\mathbf{role} \wedge \\ y.\mathbf{time} < z.\mathbf{time} < x.\mathbf{time})))$$

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:

$$\forall x(x.\mathbf{tasktype} = \mathbf{retract} \Rightarrow \\ \exists y(y.\mathbf{tasktype} = \mathbf{grant} \wedge y.\mathbf{to} = x.\mathbf{to} \wedge y.\mathbf{agent} = x.\mathbf{agent} \wedge \\ y.\mathbf{role} = x.\mathbf{role} \wedge y.\mathbf{time} < x.\mathbf{time} \wedge \\ \forall z(z.\mathbf{tasktype} = \mathbf{grant} \wedge z.\mathbf{agent} = y.\mathbf{to} \wedge z.\mathbf{role} = y.\mathbf{role} \wedge \\ y.\mathbf{time} < z.\mathbf{time} < x.\mathbf{time} \Rightarrow \\ \exists w(w.\mathbf{tasktype} = \mathbf{retract} \wedge w.\mathbf{agent} = z.\mathbf{agent} \wedge w.\mathbf{to} = z.\mathbf{to} \wedge \\ w.\mathbf{role} = z.\mathbf{role} \wedge z.\mathbf{time} < w.\mathbf{time} < x.\mathbf{time}))))$$

Rule (i). The *agent engagement principle* says that agents can only be attached to a task for a case if the agent is engaged to the case. The tasks that mark the begin and end of an engagement period have task-type “engage” and “release”. An agent can be involved in only one engagement at a time. This is expressed by the first engagement rule:

$$\begin{aligned}
& \forall x, y(((x.\mathbf{tasktype} = \mathbf{engage} \wedge y.\mathbf{tasktype} = \mathbf{engage} \wedge x.\mathbf{agent} = y.\mathbf{agent} \wedge \\
& \quad x.\mathbf{time} \leq y.\mathbf{time} \wedge x \neq y) \Rightarrow \\
& \quad \exists z(z.\mathbf{tasktype} = \mathbf{release} \wedge z.\mathbf{agent} = y.\mathbf{agent} \wedge \\
& \quad x.\mathbf{time} < z.\mathbf{time} < y.\mathbf{time} \wedge x.\mathbf{case} = z.\mathbf{case})) \\
& \quad \wedge \\
& \quad ((x.\mathbf{tasktype} = \mathbf{release} \wedge y.\mathbf{tasktype} = \mathbf{release} \wedge x.\mathbf{agent} = y.\mathbf{agent} \wedge \\
& \quad x.\mathbf{time} \leq y.\mathbf{time} \wedge x \neq y) \Rightarrow \\
& \quad \exists z(z.\mathbf{tasktype} = \mathbf{engage} \wedge z.\mathbf{agent} = y.\mathbf{agent} \wedge \\
& \quad x.\mathbf{time} < z.\mathbf{time} < y.\mathbf{time} \wedge x.\mathbf{case} = z.\mathbf{case})))
\end{aligned}$$

The second engagement rule says that a “normal” task can only be performed by an agent that was first engaged and not released for the case.

$$\begin{aligned}
& \forall x(\neg(x.\mathbf{tasktype} \in \{\mathbf{grant}, \mathbf{retract}, \mathbf{open}, \mathbf{close}, \mathbf{engage}, \mathbf{release}\}) \Rightarrow \\
& \quad \exists y(y.\mathbf{tasktype} = \mathbf{engage} \wedge y.\mathbf{agent} = x.\mathbf{agent} \wedge x.\mathbf{case} = y.\mathbf{case} \wedge \\
& \quad y.\mathbf{time} < x.\mathbf{time} \wedge \neg \exists z(z.\mathbf{tasktype} = \mathbf{release} \wedge z.\mathbf{agent} = x.\mathbf{agent} \wedge \\
& \quad x.\mathbf{case} = z.\mathbf{case} \wedge y.\mathbf{time} < z.\mathbf{time} < x.\mathbf{time}))
\end{aligned}$$

4.3 Resource balancing rules

Rule (j). 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:

$$\Sigma(y : y.\mathbf{task} = \mathbf{use}) y.\mathbf{bread} \leq \Sigma(y : y.\mathbf{task} = \mathbf{deliver}) y.\mathbf{bread}$$

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

$$\begin{aligned}
& \forall x, y(x.\mathbf{task} = \mathbf{deliver} \wedge x.\mathbf{bread} > 0 \wedge \\
& \quad y.\mathbf{task} = \mathbf{deliver} \wedge y.\mathbf{bread} > 0 \wedge x.\mathbf{case} = y.\mathbf{case} \Rightarrow x = y)
\end{aligned}$$

Rule (l). 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. So there are two sets of resources: $Res = Res_1 \cup Res_2$ and $Res_1 \cap Res_2 = \emptyset$, where Res_1 represents all the global resources and Res_2 all the local resources. In general the *resource balancing rules* make use of the formulae ψ_0 for global resource balancing and ψ_1 for local resource balancing rules:

$$\begin{aligned}
& \psi_0(\mathbf{r}) := \Sigma(y : \mathbf{true}) y.\mathbf{r}, \mathbf{r} \in Res_1 \\
& \psi_1(\mathbf{r}, c) := \Sigma(y : y.\mathbf{case} = c.\mathbf{case}) y.\mathbf{r}, \mathbf{r} \in Res_2
\end{aligned}$$

In the same way as for the task-order rules we can define a syntax:

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

Again we also require that the variable in E_3 is everywhere used in the same position 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 do not need 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
- $\forall e_1 \in \beta \forall e_2 \in \alpha - \beta (e_1(\mathbf{time}) < e_2(\mathbf{time}))$

Formally the semantics of an expressions $e \in E$ is 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 . Here a *variable binding* is defined as a partial function $\Gamma : \mathcal{X} \rightarrow \mathcal{E}$ which maps variables to events. For the full definition of the syntax and semantics the reader is referred to Appendix A.

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

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

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

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

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 the formula **true** or **false**.*

Theorem 1. φ is fs iff $\neg\varphi$ is ps.

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

Theorem 2. If φ_1 and φ_2 are fs then $\varphi_1 \vee \varphi_2$ and $\varphi_1 \wedge \varphi_2$ are fs. If φ_1 and φ_2 are ps then $\varphi_1 \vee \varphi_2$ and $\varphi_1 \wedge \varphi_2$ are ps.

Some BRL-formulas are not past-stable but it is sometimes 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 φ is checked by formula ψ if it holds that $\alpha, \emptyset \vdash \psi \rightsquigarrow 0$ iff $\forall \beta \supseteq_P \alpha (\beta, \emptyset \vdash \varphi \rightsquigarrow 0)$. If a formula is not checked by the formula **true** then it can be meaningfully monitored and therefore we say that φ 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, and so it is checked by **true**.

5.2 Monitor modeled as labeled transition system

Let us assume for a moment that we have a set of past-stable BRL-formulas. Their conjunction should hold. We 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 past stable we know that from now on the process is violating the BRL-formulas. We model this as follows. We start with an empty event $\log \epsilon$. 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 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 abstraction 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 ($p \in \mathcal{P}$)”. However there are (non trivial) subsets of BRL-formulas for which bounded abstractions of the log are sufficient. We discuss this later in this section.

We 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 $e_1, e_2, \dots \in \mathcal{E}$ with $e_{i-1}(\mathbf{time}) < e_i(\mathbf{time})$ for each $i > 0$. Let $\alpha_i = \{e_1, \dots, e_i\}$ 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;
- $\delta : Q \times \mathcal{E} \rightarrow Q$, 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 φ iff there is a function $\Psi : \{\alpha_i \mid i \geq 0\} \rightarrow Q$ with

- $\Psi(\alpha_0) = q_0$;
- $\alpha_i, \emptyset \vdash \varphi \rightsquigarrow 0 \Leftrightarrow \Psi(\alpha_i) \in F$;
- $\delta(\Psi(\alpha_i), e) = \Psi(\alpha_i \cup \{e\})$, 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 φ_1 and φ_2 are finite-state, so are $\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2$ and $\neg\varphi_1$.*

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

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

5.3 Closed and Open Cases

We further optimize some interesting special past-stable BRL-formulas. We make here the distinction between closed and open cases. A case is closed if we are sure that all the events of the case already happened. If a case is not open we call it closed. In this section we only consider well-behaved logs where for every case there is a task “open” and a possible task “close” and where no event of a case happens before its open-task or after its closed-task, i.e., where the event log satisfies rule (e) (see section 4).

We split well-behaved logs into parts that concern closed cases and parts that concern open cases. For a well-behaved log α we let α_{closed} denote that part that concerns closed cases, i.e., $\alpha_{closed} = \{ev \in \alpha \mid \exists ev' \in \alpha (ev'(\mathbf{case}) = ev(\mathbf{case}) \wedge ev'(\mathbf{task}) = \text{close})\}$ and we let α_{open} denote that part that concerns open cases, i.e., $\alpha_{open} = \alpha - \alpha_{closed}$. Based on this we can define a notion of active case formula, which captures the intuition that in order to verify a formula for a certain log we can ignore the closed cases if the formula was satisfied for these cases.

Definition 1. *We call a formula φ an active case formula if it holds for all well-behaved logs α that if $\alpha_{closed}, \emptyset \vdash \varphi \rightsquigarrow 1$ then $\alpha, \emptyset \vdash \varphi \rightsquigarrow 1$ iff $\alpha_{open}, \emptyset \vdash \varphi \rightsquigarrow 1$.*

The following theorem says that in some important BRL-formulas we can indeed delete the closed cases from α .

Theorem 6. *Let $\varphi(x)$ be a BRL-formula with one free variable x such that all quantifiers are of the form $\forall y(y.\mathbf{case} = x.\mathbf{case} \wedge \dots)$, $\exists y(y.\mathbf{case} = x.\mathbf{case} \wedge \dots)$ or $\Sigma(y : y.\mathbf{case} = x.\mathbf{case} \wedge \dots)$ then a BRL-formula of the form $\forall x(\varphi(x))$ is an active case formula.*

6 Monitor modeled as colored Petri nets

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 advantage of modeling with Petri nets over labeled transitions systems, is that the state is distributed over places and that transitions have only a local effect on the connected places. This makes it easy to refine a model by adding more structural elements, i.e., places and transitions. That is what we do in this section. We start with a colored Petri net that models the state of the labeled transition system of Section 5.2, i.e., the event log, as one token in a place called "event-log". Afterwards we refine the model to represent certain business rules by structural elements of the Petri net. The verification of the business rules is equivalent with firing a transition: each event corresponds to firing a transition in the colored Petri net and if the transition is enabled in the net, then the rules are still valid, if not then at least one rule will be violated. Note that we reduce on-the-fly auditing to simulating a colored Petri net. The resulting colored Petri net might not be very readable, however that is not our goal: it will be generated from the business rules and the monitor should be a black box for the users.

The time complexity of computing the enabling of a transition and the memory usage depend only on the number and size of tokens in the net. In many practical cases we only have business rules that can be transformed into a colored Petri nets with a bounded number of tokens of a bounded size. 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).

6.1 Colored Petri Nets

We use here colored Petri nets for modeling purposes. We refer to the standard literature for more extensive 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 $\text{dom}(\tau) = P$ and for $p \in P$: $\tau(p)$ is a type called *color set*, ν is a function with $\text{dom}(\nu) = F$ and for each $f \in F$: $\nu(f)$ is an expression called an *arc inscription*, μ is a function with $\text{dom}(\mu) = T$ and for each $t \in T$: $\mu(t)$ is a *predicate* over the variables of the arcs of the transition called a *guard* and finally m_0 is the *initial state* (initial marking). A *marking* m is a distribution of tokens over the places, where each token has a value that belongs to the color set of the place. Formally a marking is a function: $m : P \rightarrow (D \rightarrow \mathbb{N})$ where D is the set (domain) of all possible token values, i.e., $D = \bigcup_{p \in P} \tau(p)$, and $\forall (t, v) \in m(p) : v \subseteq \tau(p)$, and we say t is the token and v is its color.

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 this official syntax since we prefer to use standard mathematical notations for color sets and functions. The arc inscriptions

we consider are only *variables*. Note that all arc inscriptions per transition are unique. An arc with double arrowheads is called a bi-flow and actually it stands for two arcs: one input and one output. If x is the variable on a bi-flow then this x belongs to the input arc and we assume (but do not display) the variable x' on the output arc. The *enabling* rule is as usual: we try to find a binding for all variables on the arcs connected to the transition, such that each input variable is bound to an input token and the output variables are free but of the type of the output places. If a binding that makes the guard true exists, then the transition is enabled and may *fire* with this binding. If it fires the tokens bound to the input variables are consumed and the output variables are transformed into output tokens. We allow to produce more than one token per output variable. So the guard is actually a combination of a *pre-condition* (binding input variables to existing tokens) and a *post-condition* (binding output variables to new tokens). Note that the standard semantics of a colored Petri net is a labeled transition system. The formalization of this semantics is not trivial but well-known, see e.g. [7] and [8] for the approach.

In Fig. 1 we see a transition t with input places a , b and c and output places c and d all with type \mathbb{Q} . The guard says that t is enabled if two tokens can be found in a and b such that $x \neq y$ and the result is a (new) token in place c with value equal to the set of all input pairs so far and one new token in place d with value the sum of the input values. In Tables 1 and 2 this is displayed.

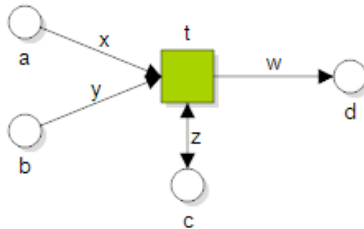


Fig. 1. A transition with arc inscriptions

Table 1. Places of Fig. 1

place	colorset
a	\mathbb{Q}
b	\mathbb{Q}
c	$\mathcal{P}(\mathbb{Q} \times \mathbb{Q})$
d	\mathbb{Q}

Table 2. Transitions of Fig. 1

transition	input	output	guard
t	x, y, z	z', w	$x \neq y \wedge z' = z \cup \{(x, y)\} \wedge w = (x + y)$

We also use the concept of a *workflow net*, a special class of Petri nets (cf. for details see [9, 10]). 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 sometimes consider colored Petri nets in which subnets have the structure of a workflow net.

6.2 Representing the labeled transition system as colored Petri net

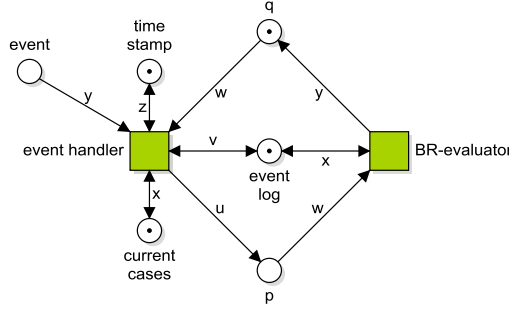


Fig. 2. The basic colored Petri net

We start with a straightforward translation of the labeled transition system of Section 5.2 into a colored Petri as shown in Fig. 2. Here we see a colored Petri net with two transitions and six places. We assume that the environment, i.e., the BIS, delivers event tokens in place “event”, but only if the place is empty! According to business rule (e) (see section 4) we require that each case starts with a task with type “open” and has no more events after a task with type “close”. Transition “event handler” consumes tokens from the input place “event”. There are two guards: the time stamp $e.time$ of a new event e should be greater than as the time in place “time stamp” and the case identifier $e.case$ must occur in the token from place “current cases” unless the task $e.tasktype \in \{open, grant, retract\}$. If the task is $e.tasktype = open$ then the $e.case$ is inserted in the token from “current cases”. So the “event handler” is the gate keeper. If the “event handler” fires the event e is inserted into the token in the “event log” and a colorless (black) token is put in place p . This token is the trigger for transition “BR-evaluator” which has a black token as output for place q . It has a guard which is the conjunction of all business rules. Since the

business rules are decidable the guard can be computed. After a token is put in the place “event”, the marking displayed in Fig. 2 should be reachable. If so, then the business rules are not violated.

Table 3. Places of Fig. 2

place	colorset
event	E
current cases	2^C
time stamp	\mathbb{Q}
p	$2^\emptyset, (black)$
q	$2^\emptyset, (black)$
event log	2^E

Table 4. Transitions of Fig. 2

transition	input	output	guard
event handler	x, y, z, v, w	x', z', v', u	ϕ
BR-evaluator	x, w	x', y	$x \models BR \wedge w = \emptyset \wedge y = w$

Note that colorset 2^\emptyset only contains the value \emptyset which represents the “black” token of classical Petri nets.

Here BR stands for the conjunction of all business rules and the guard evaluates them. Only if there is a binding in the event log, i.e., variable v , that makes it “true” then the transition will fire.

$$\begin{aligned} \phi &= z < y.time \wedge (\neg(y.task \in \{open, grant, retract\}) \Rightarrow y.case \in x) \wedge \\ &v' = v \cup \{y.case\} \wedge u = \emptyset \wedge t.task = open \Rightarrow x' = x \cup \{y.case\} \wedge \\ &y.task = close \Rightarrow x' = x \setminus \{y.case\} \wedge (t.task \in \{open, close\} \vee x' = x) \end{aligned}$$

Theorem 7. *In the colored Petri net of Fig. 2 the marking with no tokens in places event and p and one token in q is reachable from the initial marking if and only if the business rules are valid until then.*

Note that the colored Petri net of Fig. 2 is not the one we want since the token in “event log” grows unlimited, so verification will take increasing time. Therefore we restrict ourselves to a subset of the business rules and for this set we refine the colored Petri net in the sense that we add more structural Petri net elements (places and transitions) that replaces several formulae in the guard of the BR-evaluator and it will reduce the size of the log.

6.3 First refinement: resource balancing and authorization rules

In the first refinement we assume that we only have the following types of formulae:

- resource balancing rule (l) of Section 4.3.
- authorization rules (f),(g),(h) and (i) of Section 4.2.
- arbitrary “active case rules”

In this refinement we still have an event log, but restricted to events belonging to the *current* cases, i.e., cases that have been opened and not yet closed. Later on we restrict the task-order rules and then we develop a second refinement in which we do not need a place that represents an event log any more. The first refined model is displayed in Fig. 3 and is a refinement where the authorization rules and the resource balancing rules are replaced by structural elements. The BR-evaluator is now replaced by a transition called “case rule evaluator” which only works on the “current cases”. In order to understand this model we first model the resource balancing and authorization rules in isolation.

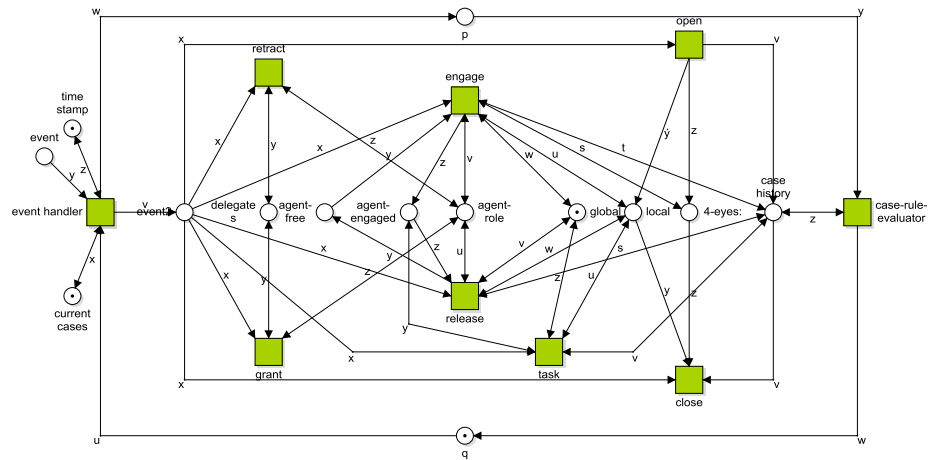


Fig. 3. The first refinement

We now explain the first refinement model, displayed in Fig. 3. We add places and transitions. Actually the tokens in these places can be seen as abstractions from the event log. The refined model is constructed from the basic model in Fig. 2 by adding representations of the different rules: the resource balancing rules, the delegation rules, the four-eyes principle and the agent engagement rules in that order. This is explained in the following paragraphs. We do not distinguish different tasks, but only different task-types. Note that we do not specify the transition guards in detail, because they depend on the specific business rules. We start with the resource balancing rules. The color sets of the places are either the ones defined for

Resource balancing rules. There are two kinds of resource balancing rules, global and local resource balancing rules. They will be modeled by embedding Fig. 4 (i) into Fig. 2. We create one place for all the global resources and one for all the local resources, in Fig. 4 (i) called “global” and “local”. The color set of “global” is a finite subset of $Res_1 \rightarrow \mathbb{Q}$ and the color set of “local” is a finite subset of $C \times (Res_2 \rightarrow \mathbb{Q})$, where $Res = Res_1 \cup Res_2$ and $Res_1 \cap Res_2 = \emptyset$, where Res_1 represents all the global resources and Res_2 all the local resources. The global resource place contains always one token, representing the current status of all the global resources. In the initial state it has the initial values of the resources. Each transition is allowed to manipulate the resources: by additions or subtractions. So we connect each case task to these two resources places with an input and an output arc so that each transition can update the resources. The local resource place has type $C \times (Res_2 \rightarrow \mathbb{Q})$. For the local resource place we have two more connections: one input arc from the “open” transition and one output arc to the “close” transition. This is because the local resources are case-specific and as soon as the case is closed this token is not necessary any more. The “open” transition creates for a new case one token in the “local res” with an initial value for each local resource. Now we have modeled the updating of the resources. The only thing that remains is the modeling of the resource balancing rules. We take them out of the guard of the “BR-evaluator” and put them in the guard of the case task transitions, because they are the only ones who could make them invalid.

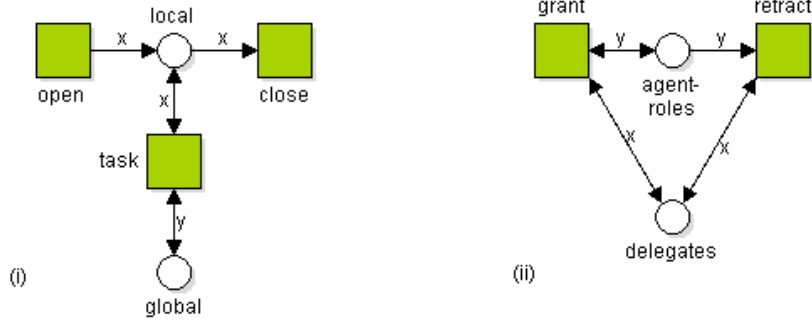


Fig. 4. Resource rules (i) and delegation rules (ii)

Next we consider the *authorization* rules.

Delegation rules. The delegation rules will be modeled by embedding Fig. 4 (ii) into Fig. 2. In Fig. 4 (ii) we see a subnet of Fig. 3 with two tasks “grant” and “retract” and two places “agent-roles” and “delegates” with one token each with color sets $2^{A \times R \times A}$ and $2^{A \times R \times N}$ respectively. These sets are always bounded

since we assume the sets A and R to be fixed. The meaning of this is that in “agent roles” the token is a set containing for each role an agent and the agent from whom it was obtained in the current state. In “delegates” the token is a set that contains for each role an agent together with the amount of delegates it has created for this role.

Table 5. The guard of of “grant” in Fig. 4 (ii)

guard of grant
$(b, r, c) \in y \wedge y' = y \cup \{(a, r, b)\} \wedge (b, r, n) \in x \wedge x' = (x \setminus \{(b, r, n)\}) \cup \{(b, r, n + 1)\}$

Table 6. The guard of “retract” in Fig. 4 (ii)

guard of retract
$(a, r, b) \in y \wedge \{(b, r, n), (a, r, 0)\} \subseteq x \wedge x' = (x \setminus \{(b, r, n)\}) \cup \{(b, r, n - 1)\}$

The guards explain the working: if “grant” fires it adds a triple with value (a, r, b) meaning that agent a receives from agent b the role r . (Note that the a is coming from another input that we did not display here). In order to be able to do so the triple (b, r, c) should be available, meaning that agent b has been granted role r by some agent c . The pre-condition is that the triple (b, r, n) is in the token of place “delegates” meaning that agent b has granted role r to other agents already n times. Finally the token is update by replacing the triple by $(b, r, n + 1)$. 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 the token of 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 expresses that the agent in that role is needed for the task. Note that we assume that an agent can have more than one role at the same time.

Four-eyes principle. Next we show how the *four-eyes principle* can be represented. We generalize the four eyes principle in the sense that there can be a set of tasks for which the all agents should be different per case. In order to model this we added one place called “4-eyes” (see Fig. 3) with color set $C \times 2^{T \times A}$. So the place stores, per current case, the set of all pairs of a task and an agent that occurred in a case so far. This place is connected to transition “engage” with a bi-flow in order to enable this transition to update the token. Transition “open” is input for this place and transition “close” is output for it, they create and destroy the token per case. The business rule that checks the four-eyes principle

is a guard for the transition “engage”, because this transition is the only one that can make the rule invalid.

Agent engagement rules. Next we consider the *agent engagement* rules. These rules say that an agent can be engaged in only one case at a time. So in order to engage an agent it should be free. We model this in Fig. 3 with the places “agent-free” and “agent-engaged”, with color sets A and $A \times C$. Initially the first one contains all agents and the second one is empty. In the “agent-engaged” place we record to which case the agent is engaged. These places are connected to transitions “engage” and “release” in the obvious way: “agent-engaged” is input for “release” and output for “engage” and similarly “agent-free” is input for “engage” and output for “release”. For all other tasks there is one transition “normal-task” that checks whether the agent it needs is engaged for the case and which updates the “case-history” like all other transitions.

Finally we note that we have restricted the other business rules to *single-case* rules, which means that we only need the current cases to check them. That is done in Fig. 3 by transition “case-rule-evaluator” using the place “case-history”. Per current case there is one token in this place, created by “open” and removed by “close”. the transitions “engage” and “release” update these tokens. So we may formulate the following theorem.

Theorem 8. *In the colored Petri net of Fig. 3 the final marking with no tokens in places event and p and one token in place q is reachable from the initial marking if and only if the business rules are valid until then.*

We summarize the places and their color sets of the model of Fig. 3

Table 7. Places of Fig. 3

place	colorset
event	E
event2	E
current cases	2^C
time stamp	\mathbb{Q}
p	$2^0(\text{black})$
q	$2^0(\text{black})$
delegates	$2^{A \times R \times A}$
agent-roles	$2^{A \times R \times N}$
global	$Res_1 \rightarrow \mathbb{Q}$
local	$C \times (Res_2 \rightarrow \mathbb{Q})$
agent-free	A
agent-engaged	$A \times C$
4-eyes	$C \times 2^{T \times A}$
case history	2^E

6.4 Second refinement: task-order rules

In the former refinement we had for each task-type a transition. In the second refinement we introduce transitions for all the tasks and places for the basic counting formula. The model is displayed in Fig. 5. The color sets are as in Table 7.

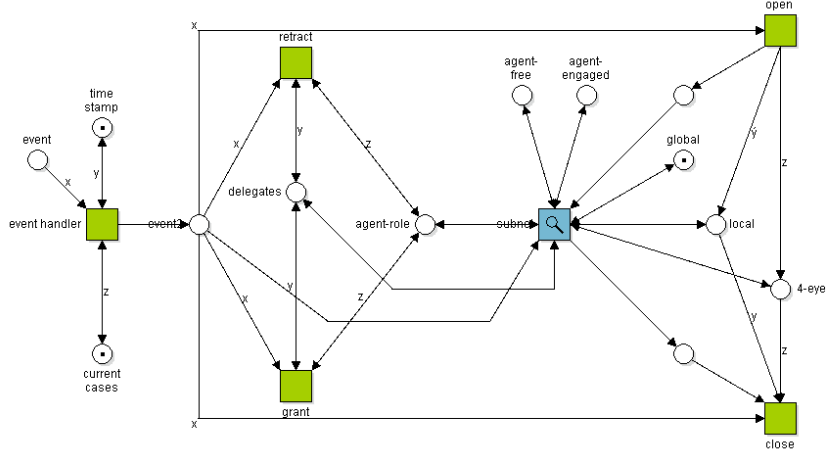


Fig. 5. Expressing the task-order rules

In this section we consider a refinement for task-order rules. We assume that the task-order rules are only formulated as (a conjunction of) *basic counting formula* as defined by rule (c) in Section 4.1. If we restrict the task-order rules in this way we can remove from Fig. 3 the place “event-history” and transition “case-rule-evaluator” by adding more structure. The transition in Fig. 5, with name “subnet” and displayed with an eye glass symbol, will be filled by a task-order specific subnet. As an example consider the counting formula:

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

where a, b, d, e, g are tasks. As explained in [6] each such a rule puts a restriction on the firing of transitions which can be expressed by a place.

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 right-hand side of the \geq sign. In Fig. 6 we see the transformation of $\forall c(\phi(a, c) + \phi(b, c) \geq \phi(d, c) + \phi(e, c))$ into the Petri net with one place called p and four transitions $\{a, b, d, e\}$.

We may reduce such a business rule by eliminating *implicit places* (cf. [11]). Instead of using such standard Petri nets techniques for this elimination we can eliminate basic counting formula. Reduction means that if we have a subformula

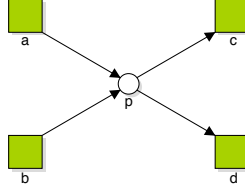


Fig. 6. Transformation of a basic counting formula

φ in the conjunction that is implied by one or more other subformula, then we delete φ . 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 $\forall c(\phi(a, c) + \phi(b, c) \geq \phi(d, c) \wedge \phi(b, c) \geq \phi(d, c) + \phi(e, c))$ then we delete $\phi(a, c) + \phi(b, c) \geq \phi(d, c)$. 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 right-hand side of the \geq sign. We assume that the task-order rules are in the normal form described above and are irreducible.

Note that in the language-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. [12, 13]).

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 Petri net. Note that since we consider only the conjunction of business rules we may combine the task-order rules into one business rule. As an example consider the formula:

$$\begin{aligned} \forall c(\phi(open, c) \geq \phi(a, c) + \phi(g, c) \wedge \phi(a, c) + \phi(f, c) \geq \phi(b, c) \wedge \\ \phi(b, c) \geq \phi(d, c) \wedge \phi(d, c) \geq \phi(e, c) + \phi(f, c) \wedge \\ \phi(e, c) + \phi(q, c) \geq \phi(close, c) \wedge \\ \phi(g, c) \geq \phi(h, c) \wedge \phi(h, c) \geq \phi(j, c) \wedge \phi(h, c) \geq \phi(k, c) \wedge \\ \phi(j, c) \geq \phi(l, c) \wedge \phi(k, c) \geq \phi(m, c) \wedge \phi(l, c) \geq \phi(n, c) \wedge \\ \phi(m, c) \geq \phi(n, c) \wedge \phi(n, c) \geq \phi(q, c)) \end{aligned}$$

Applying the construction rules we obtain the (classical) Petri net in Fig. 7. This classical Petri net is a *workflow* net and is *sound* (cf. [10]). This means that transition “close” always eventually can fire and that no tokens are left if it fires.

In case the task-order rules do not determine a sound workflow net we still can transform them into a Petri net. In Fig. 8 we show this construction for the simple case where the rule of Fig. 6 is the only task-order rule. Here we considered only one basic counting formula, namely the same as in Fig. 6, but the construction for more is obvious. We use here a *reset arc* (cf. [11]) to all places introduced by the basic counting formula and the “close” transition. A transition removes all tokens from the places its connected to by a reset arc. This also guarantees that after firing “close” no tokens are left. In Fig. 8 the arc between place p and transition “close” is a reset arc.

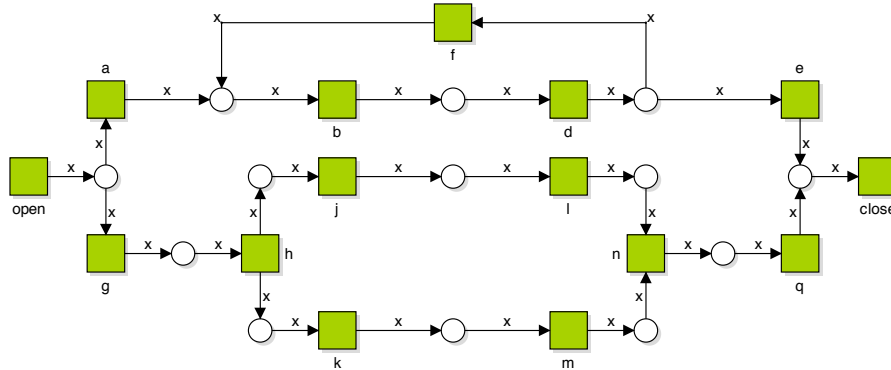


Fig. 7. Example of a workflow net with two case types

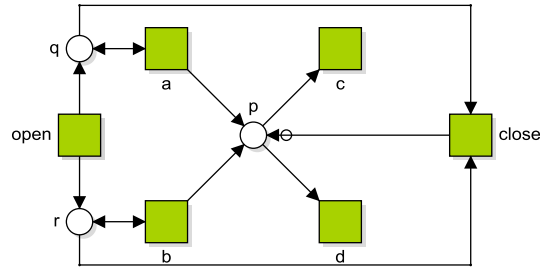


Fig. 8. A Petri net for non-workflow situations

Adding color to the classical subnet. So far we translated the task-order rules into a classical Petri net. Next we augment our net with color and then we embed it into the net of Fig. 5:

- All places in this net obtain the case identifiers, i.e., C , as color set.
- All transitions have arc inscriptions with the same variable (from C) which means that transitions can only fire if they consume tokens with the same case identifier and that they produce tokens with the same case identifier.
- Reset arcs will be color sensitive, i.e., they will only remove tokens with the same value (case identifier) as one of the other inputs. The guard requires that there is only one case identifier in the other input.
- Transitions of the subnet are connected to the places in Fig. 5 as is done for transitions in Fig. 3 with the corresponding task type.

Expressing this for a colored Petri net tool like CPN-tools (cf. [7]) is straightforward.

Theorem 9. *The task-order rules are valid if and only if the firing sequence from the event log is executable on the constructed Petri net.*

Since we have now translated both the task-order rules as well as the resource balancing and authorization rules we have the following corollary.

Corollary 2. *For all translated rules it holds that they are valid if and only if the firing sequence from the event log is executable on the constructed Petri net.*

6.5 Limits on memory usage and computing time

In this subsection we show that the token in “case history” is *probabilistically* bounded. We make the following assumptions:

- new case arrive with a frequency λ
- the time between the arrival of an “open” event and a “close” event for a case is on average w
- the number of events of case i is a random variable M_i . The expected value of M_i is $\mathbb{E}[M_i] = m$
- the random variables M_i are independent and identically distributed and also independent of L .

The amount of memory needed and the evaluation time of guards is a proportional to the amount of events stored. So we have to determine a bound on the number of events in the model.

Theorem 10. *Let L be a random variable expressing the number of active cases in the stable situation, let M_i be the random number of events of case i in the active cases. Then the probability that the number of cases in the case history exceeds b is bounded by:*

$$\mathbb{P}\left[\sum_{i=1}^L M_i \geq b\right] \leq l.m/b$$

Proof. By Little’s formula the expected value of L is $\mathbb{E}[L] = \lambda.w$. By Markov’s inequality we have: $\mathbb{P}[\sum_{i=1}^L M_i \geq b] \leq \mathbb{E}[\sum_{i=1}^L M_i]/b$. By Wald’s formula we have $\mathbb{E}[\sum_{i=1}^L M_i] = \mathbb{E}[L].m$. For a derivation of the used formula see [14]. \square

So, if we accept a memory overflow with probability α then we have a bound $b = l.m/\alpha$.

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 cannot trust the BIS. We have shown how to develop a monitor either as labeled transition system with an infinite state space, or as a colored Petri net with tokens that grow unboundedly in size. In order to be able to evaluate business rules in such a way that we need only a finite memory and that the computations involved

will not depend on the size of the event log, we restricted ourselves to a subset of the BRL. This subset covers all the characteristic examples we encountered in practice. The construction method we presented is based on two principles: for the authorization rules and resource balancing rules we have given a fixed colored Petri net where the specific parameters for a particular organization have to be filled in by token values, e.g., for the four eyes principle. For the task-order rules we presented a construction method to extend the colored Petri net in a systematic way. We do not claim that our constructions are unique or the best, but they seem to be rather straightforward. We have shown that the involved computations are often not depending on the length of the event log. For active case rules we have shown probabilistic bounds on the number of cases that need to be verified. If the Petri net cannot execute a transition, then a rule is violated. This violation can be reported or it may generate an interrupt for the BIS. As future work we will try to find a larger subset of business rules that can be translated to colored Petri net patterns that can be glued together to form one colored Petri net in a systematic way. We are involved in a case study in practice using an implementation based on CPN tools (cf. [15]) which shows that the approach is feasible. As future work we also see a more detailed analysis of past stable formulas in BRL and there is a need for a more user friendly form for BRL, by sugaring the syntax or even a graphical language. Last but not least we note that an approach is needed to recover from a violation of the business rules.

References

1. Romney, M.B., Steinbart, P.J.: Accounting Information Systems. 11 edn. Pearson International Editions (2009)
2. Geerts, G.L., McCarthy, W.E.: An ontological analysis of the economic primitives of the extended-REA enterprise information architecture. *International Journal of Accounting Information Systems* **3**(1) (2002) 1–16
3. McCarthy, W.E.: The REA accounting model: A generalized framework for accounting systems in a shared data environment. *The Accounting Review* **57**(3) (1982) 554–578
4. Berg, D.: Turning Sarbanes-Oxley projects into strategic business processes. *Sarbanes-Oxley Compliance Journal* (2004)
5. IFRS Foundation: The International Accounting Standards Board Web Site. <http://www.ifrs.org/Home.htm> (2010)
6. van Hee, K., Serebrenik, A., Sidorova, N., van der Aalst, W.: History-dependent Petri nets. In: ICATPN’07: Proceedings of the 28th international conference on Applications and theory of Petri nets and other models of concurrency. Volume 4546 of LNCS., Berlin, Heidelberg, Springer-Verlag (2007) 164–183
7. Jensen, K.: Coloured Petri nets: basic concepts, analysis methods and practical use, vol. 2. Springer-Verlag, London, UK (1995)
8. van Hee, K.M.: Information Systems Engineering, a Formal Approach. Cambridge University Press, New York, NY, USA (1994)
9. van der Aalst, W.M.P., van Hee, K.M.: Workflow Management: Models, Methods and Systems. MIT Press (2002)

10. van der Aalst, W.M.P.: Verification of workflow nets. In: ICATPN 1997. Volume 1248 of LNCS., London, UK, Springer-Verlag (1997) 407–426
11. Girault, C., Valk, R.: Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2001)
12. Darondeau, P.: Deriving Unbounded Petri Nets from Formal Languages. In: CONCUR 1998, London, UK, Springer-Verlag (1998) 533–548
13. Werf, J., Dongen, B., Hurkens, C., Serebrenik, A.: Process Discovery using Integer Linear Programming. In van Hee, K.M., Valk, R., eds.: ATPN. Volume 5062 of LNCS., Springer (2008) 368 – 387
14. Ross, S.: Introduction to Probability Models. Academic Press (2007)
15. Jensen, K., Kristensen, L.: Coloured Petri nets : modelling and validation of concurrent systems. Springer (2009)

A The Formal Syntax and Semantics of BRL

We first recall the syntax of BRL which is defined by the following abstract syntax:

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

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, \dots range over expressions in E , and the variable τ ranges over the types in Θ .

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

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

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

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

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

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 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 $\Gamma : \mathcal{X} \rightarrow \mathcal{E}$. 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 event 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:

$$\begin{array}{c}
\frac{\alpha, \Gamma \vdash e \rightsquigarrow b \quad b \in \{0, 1\}}{\alpha, \Gamma \vdash \neg e \rightsquigarrow (1 - b)} \\
\\
\frac{\alpha, \Gamma \vdash e_1 \rightsquigarrow b_1 \quad \alpha, \Gamma \vdash e_2 \rightsquigarrow b_2 \quad \{b_1, b_2\} \subseteq \{0, 1\}}{\alpha, \Gamma \vdash (e_1 \wedge e_2) \rightsquigarrow (b_1 \cdot b_2)} \quad \frac{(p, v) \in \Gamma(x)}{\alpha, \Gamma \vdash x.p \rightsquigarrow v} \\
\\
\frac{b \in \{0, 1\} \quad (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\} \quad (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\} \quad (b = 1) \Leftrightarrow (\Gamma(x_1) = \Gamma(x_2))}{\alpha, \Gamma \vdash (x_1 = x_2) \rightsquigarrow b} \quad \frac{q \in \mathbb{Q}}{\alpha, \Gamma \vdash q \rightsquigarrow q} \quad \frac{a \in A}{\alpha, \Gamma \vdash a \rightsquigarrow a} \\
\\
\frac{r \in R}{\alpha, \Gamma \vdash r \rightsquigarrow r} \quad \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 \quad \alpha, \Gamma \vdash e_2 \rightsquigarrow v_2 \quad \{v_1, v_2\} \subseteq \mathbb{Q}}{\alpha, \Gamma \vdash (e_1 + e_2) \rightsquigarrow (v_1 + v_2)} \quad \frac{\alpha, \Gamma \vdash e_1 \rightsquigarrow v_1 \quad \alpha, \Gamma \vdash e_2 \rightsquigarrow v_2 \quad \{v_1, v_2\} \subseteq \mathbb{Q}}{\alpha, \Gamma \vdash (e_1 \cdot e_2) \rightsquigarrow (v_1 \cdot v_2)}
\end{array}$$

It can be observed that for each well-typed expression e such that $e : \tau$ it holds for every event 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 $v \in \llbracket \tau \rrbracket$. 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 $f : W \rightarrow V$ 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 event 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.