A Framework for Data-Driven Workflow Management: Modeling, Verification and Execution

Nahla Haddar, Mohamed Tmar, and Faiez Gargouri

University of Sfax, B.P. 1069, 3029 Sfax, Tunisia
`nhaddar@ymail.com`, {`mohamed.tmar,faiez.gargouri`}`@isimsf.rnu.tn`

Abstract. In recent years, many data-driven workflow modeling approaches has been developed, but none of them can insure data integration, process verification and automatic data-driven execution in a comprehensive way. Based on these needs, we introduced, in previous works, a data-driven approach for workflow modeling and execution. In this paper, we extend our approach to ensure a correct definition and execution of our workflow model, and we implement this extension in our Framework *Opus*.

Keywords: Data-driven workflow management Framework, Petri nets, Relational algebra, Workflow analysis and verification, Soundness property.

1 Introduction

In a competitive environment continually evolving, companies are recognized the need to manage their business processes in order to align their information systems, more and more quickly, in a process-oriented way. In this context, workflow management systems (WMS) offer promising perspectives for modeling, processing and controlling processes. In the most common WMS, only the control flow [1] is completely included [1]. In Fact, during process execution, a process-oriented view (e.g. worklists) is provided to end-users. However, the behavior of an activity during its execution is out of the control of the WMS [2]. As almost all processes are related to data, such as the costs of the ordered products, the addresses information for delivery, etc., the main goal from using a WMS is to automate, as possible, the manipulation of data in business processes, and to restrict as possible the manual tasks performed by human actors.

Regarding the existing literature, the need for modeling processes that combine data and control flow has been widely studied. Most of them are inspired from Petri nets (P-nets) formalism, such as the approaches proposed in [3–7]. Thus, to enhance earlier approaches that have mainly focused on process activities and largely overlooked the data, we previously extended, in [8,9], the P-nets

---

[1] Control flow: is a set of synchronized activities representing the business process functions, and a set of ordering constraints defining their execution sequence [1].

H. Decker et al. (Eds.): DEXA 2013, Part I, LNCS 8055, pp. 239–253, 2013.

formalism by data operations inspired from the relational algebra to model data-driven workflows. This extension improves the generation of business functions from the process definition, without the need for a programmer, and provides advanced abilities for the information system (IS) and the users' integration. In this paper, we extend the modeling method of the proposed approach [8, 9] by some rules, to ensure the consistency of the process model during runtime. We also provide a technique to verify a released notion of the classical soundness property.

The remainder of the paper is organized as follows: we present the related work in Sect. 2 and we continue by introducing our workflow model in Sect. 3. In Sect. 4, we present the definition of the firing rules that ensure a uniform execution of all the process activities. Then we discuss, in Sect. 5, the technique provided to ensure the analysis and the verification of our workflow model. In Sect. 6, we present our Framework *Opus*. Section 7 concludes.

2 Related Work

The need for modeling processes integrating data has been recognized by several authors [3–7, 10, 11]. The Case Handling Paradigm [11] aims to coordinate activities which are presented as forms in relation to atomic data elements. The problem here is that data may be omitted or activities are unwittingly ignored or executed many times. In addition, more than one user can handle the same case simultaneously, which damages the data coherence.

The PHILharmonicFlows system [2] provides a comprehensive approach that combines object behavior based on states with data-driven process execution. In fact, it allows the control of activities by presenting them as form-based [2] and black-box activities [3]. The proper execution as well as termination of processes at runtime is further ensured by a set of correctness rules [12]. But, the execution of form-based activities increases the rate of errors that may be caused by the manual seizure performed by human actors, even if the seized data values respect the data types requested by the form' input fields.

Many extensions of P-nets in which tokens carry data have been defined in the literature. The workflow nets based on colored P-nets (WFCP-nets) [10] consider a P-net color as an abstraction of data objects and flow control variants. The execution of a WFCP-net depends on the interpretation of its arc expressions and guard expressions, which describe the business rules. Besides, the verification methods of workflow nets [13] are adapted to WFCP-nets. The weakness of this approach is that the process specification consists of a graphical part and a WF script part. The latter is a hard-coding process logic that describes the data elements and the behavior of activities. So, resulting applications are both complex to design and costly to deploy, and even simple process changes require

---

[2] Form-based activities: provide input fields for writing and data fields for reading selected attributes of data object instances [2].

[3] Black-box activities: allow the integration of advanced functionalities (e.g. sending e-mails) [2].

costly code adaptations and testing efforts. Another extension of P-nets is the workflow nets with data (WFD-nets) [7], in which transitions can read from or write to some data elements. This extension does not provide a support for executing process models. However, it defines algorithms to verify a soundness property that guarantees the proper termination of a WFD-net and that only certain transitions are not dead.

None of the existent approaches considers data integration, process verification and data-driven execution issues in a comprehensive way. Thus, in this area a comprehensive approach for supporting these three issues is still missing.

3 Our Data-Driven Workflow Model

As described by the most modeling approaches, a process is defined, in a higher level of abstraction, as a set of synchronized activities performed by roles according to the available data. If we stop at this level, we will not be able to generate the process functions from the process model definition, and activities will behave as a black box in which data are managed by invoked application components. To attempt the lowest level of abstraction, we propose to split each activity, in a process, to tasks applied on data. Each task consumes data to produce others. Thus, each activity is presented as a set of data-driven tasks. Each task consumed data provenance can be either the IS, or data produced by other tasks. But, in some cases, to complete the processing of a task, data can be seized by a role. Accordingly, to enable tasks to generate new data from old ones and import data from the IS, data have to be well structured. We introduced this approach in [8,9], in a formal way, as a data-driven modeling approach based on combination between structured tokens P-nets and relational algebra.

3.1 Data Structure

According to [8,9], we define each handled data as a data structure; i.e, a pair $s = (C,\ D)$, where C is a list of attributes and D is a list of data tuples.
Each tuple is an ordered list of attributes values, formally defined by:

$$C = (c_1, c_2 \dots c_n),\ D = \{(d_{1_1}, d_{1_2} \dots d_{1_n}), (d_{2_1}, d_{2_2} \dots d_{2_n}) \dots (d_{m_1}, d_{m_2} \dots d_{m_n})\} .$$

Where n (resp. m) is the number of attributes (resp. tuples) in s.

Each attribute $c_j = (\alpha_j,\ \beta_j)$ is a pair characterized by an attribute identifier α_j and an attribute type β_j, such as: $\forall j \in \{1,\ 2 \dots n\}$, $\beta_j \in \{Int,\ Float,\ Char,\ String,\ Date,\ Boolean \dots\}$, and $\forall i \in \{1, 2 \dots m\}$, an attribute value d_{i_j} is a specific valid value for the type β_j of the attribute c_j.

At modeling step, the designer has just to define the data structure attributes and the values types put up with each one. At runtime, the different data structure tuples comprise varying values according to each attribute type.

3.2 Process Structure

The workflow process is defined as a P-net representing the work, where a place corresponds to a *data structure* that contains *structured tokens* (tuples) and a transition corresponds to a *task*. A workflow is then a quadruplet [8, 9] $WF = (S,\ T,\ Pre,\ Post)$, where:

$S = \{s_1,\ s_2 \dots s_{|S|}\}$ is a finite set of data structures,

$T = \{t_1,\ t_2 \dots t_{|T|}\}$ is a finite set of tasks inspired from the relational algebra,

$Pre:\ S \times T \rightarrow \mathbb{N}$ is the pre-incidence matrix, such as, $\forall i \in \{1,\ 2 \dots |S|\}$ and $j\ \in \{1,\ 2 \dots |T|\}$, $Pre(s_i,\ t_j)$, is the edge between a data structure s_i and a task t_j weight, representing the number of tokens consumed by t_j in order to be firable, i.e. executable.

$Post:\ T \times S \rightarrow \mathbb{N}$ is the post-incidence matrix. Due to the dynamic of the relational algebra, we cannot be limited to a static post-incidence matrix thus, $\forall i \in \{1,\ 2 \dots |S|\}$ and $j \in \{1,\ 2 \dots |T|\}$, $Post(t_j,\ s_i) \in [Post_{Min}(t_j,\ s_i),$ $Post_{Max}(t_j,\ s_i)]$. Where: $Post_{Min}(t_j,\ s_i)$ (resp. $Post_{Max}(t_j,\ s_i)$), is the edge between a task t_j and a data structure s_i minimal (resp. maximal) weight, representing the minimal (resp. maximal) number of t_j produced tokens.

3.3 Data Operations

A task can be viewed as a data operation applied on data structure tokens to produce others. Therefore, we have inspired from the relational algebra to define the behavior of operations. We presented these operations in [9]. So, in this paper, we detail in Appendix *A*, only operations that we will use to demonstrate the new extensions of the model. Noting that the definition of the *Add_Tuples* operation, presented in Table 1, is an extended version of its definition in [9]. In fact, in this version, we allow to inserts all a data structure tuples in another data structure, instead of inserting only a single tuple [9].

3.4 Workflow Example

The customer solvency check role (SCRole) evaluates the received orders, and sends them to the inventory check. After the evaluation, either an order is rejected, or sent to shipping and billing. As illustrated by Fig. 1, considering that s_8, s_{13}, and s_{19} present tables from the IS, we restrict our example to the inventory check role (ICRole) sub-process, which performs the following activities:

Select the ordered products: t_7 extends s_8 (which contains all the products data) by the *ord_qtity* attribute, in order to allow the ICRole to enter the ordered quantities. Then, t_8 selects from s_9 only tuples having an ordered quantity value higher than zero and lower than the stocked product quantity.

Verify the products availability: t_9 checks s_{10} content. If it contains one or more tokens, t_9 will reproduce s_{11} token in s_{12}, otherwise, it will end the process.

Create a new order: according to the decision of t_9, if there are available products, t_{10} will add a new order in s_{13}.

Create the new order lines: the ICRole enters the new order identifier and t_{13} saves it in s_{17}. Then, t_{14} will create the new order lines and finally, t_{15} will save the resulting structured s_{18} tokens in s_{19}.

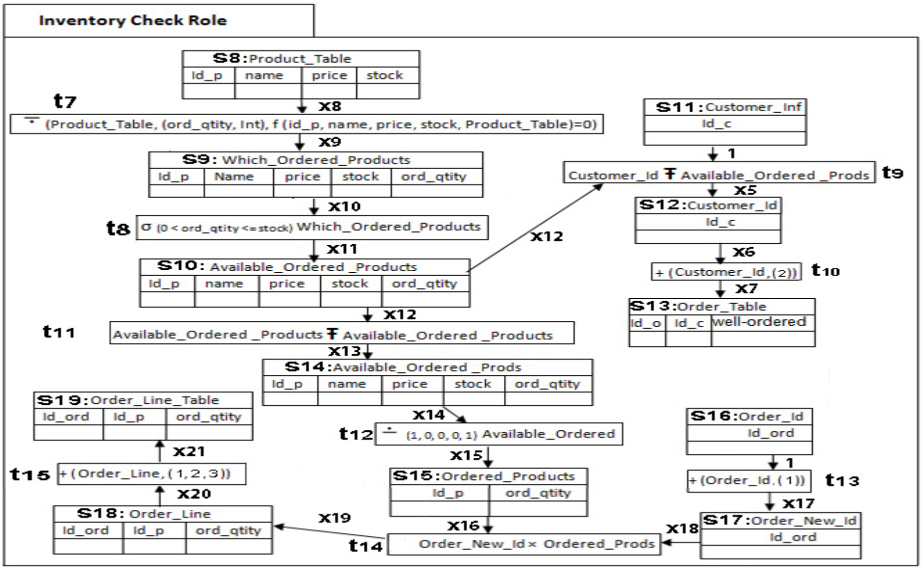


Fig. 1. The inventory check role sub-process [9] (modified version)

We deduce the *Pre* and *Post* matrix of the example in Fig. 1.

$$
Pre = \begin{array}{c} \\ s_8 \\ s_9 \\ s_{10} \\ s_{11} \\ s_{12} \\ s_{13} \\ s_{14} \\ s_{15} \\ s_{16} \\ s_{17} \\ s_{18} \\ s_{19} \end{array}
\begin{array}{c}
\begin{array}{ccccccccc} t_7 & t_8 & t_9 & t_{10} & t_{11} & t_{12} & t_{13} & t_{14} & t_{15} \end{array} \\
\begin{pmatrix}
x_8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & x_{10} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & x_{12} & 0 & x_{12} & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & x_6 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & x_{14} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & x_{16} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & x_{18} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_{20} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\end{array}
$$

Following the definition of tasks in Appendix A, the number of tokens produced by each task in Fig. 1, are defined as follows: $x_5 \in [0,\ 1]$, $x_6 = 1$ (i.e. there is only a single customer identifier), $x_7 = |D_{13}| + x_6$, $x_9 = x_8$, $x_{11} \in [0,\ x_{10}]$, $x_{13} \in [0,\ x_{12}]$, $x_{15} \in [0,\ x_{14}]$, $x_{17} = |D_{17}|+1 = 1$ (because $D_{17} = \varnothing$), $x_{19} = x_{16} \times x_{18}$, $x_{21} = |D_{19}| + x_{20}$. Accordingly, the *Post* matrix is defined by:

$$
Post = \begin{array}{c} \\ s_8 \\ s_9 \\ s_{10} \\ s_{11} \\ s_{12} \\ s_{13} \\ s_{14} \\ s_{15} \\ s_{16} \\ s_{17} \\ s_{18} \\ s_{19} \end{array}
\begin{array}{c}
\begin{array}{ccccccccc} t_7 & t_8 & t_9 & t_{10} & t_{11} & t_{12} & t_{13} & t_{14} & t_{15} \end{array} \\
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
x_8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & [0,\ x_{10}] & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & [0,\ 1] & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & |D_{13}| + x_6 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & [0,\ x_{12}] & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & [0,\ x_{14}] & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & x_{16} \times x_{18} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & |D_{19}| + x_{20}
\end{pmatrix}
\end{array}
$$

3.5 Marking

The marking $M^{[\theta]}$ defines the state of the process described by WF at a given time $\theta \in \{0,\ 1,\ 2\ldots\}$. Thus, $\forall i \in \{1,\ 2\ldots|S|\} : M^{[\theta]} = \left(m_1^{[\theta]}\ m_2^{[\theta]}\ \ldots\ m_{|S|}^{[\theta]}\right)$, where $m_i^{[\theta]} \in \mathbb{N}$ is the number of tuples in s_i.
The initial marking $M^{[0]}$ defines the state of WF at time $\theta = 0$, in which only root nodes of a WF process can be initiated by a finite number of tokens. The evolution of the markings, in the other nodes, results due to the firing of WF tasks. A valid initial marking must follow (1) [8, 9].

$$\forall j \in \{1,2\ldots|S|\}, m_j^{[0]} = \begin{cases} \max\limits_{k\in\{1,\ 2\ldots|T|\}} Pre(s_j,\ t_k) \\ \quad if\ \forall l \in \{1,2\ldots|T|\} Post_{Max}(s_j,\ t_l) = 0, \\ 0 \text{ otherwise.} \end{cases} \tag{1}$$

3.6 Synthesis

Our proposed data-driven approach allows for a comprehensive integration between data flow and control flow, which ensures a successful data driven execution of the workflow. Indeed, data integration is granted through data structures that can handle various data elements types. Furthermore, data manipulation is enable through data operations that can read, write and generate new data elements without any risk of simulating a WF process in which data can be lost. This is granted through the dynamic behavior of the relational data operations, which entails a generalization of the static post-incidence matrix of the classical P-nets. In the next section, we present how we improve our approach by the application of some firing rules. These latter grant a uniform definition of a WF process and introduce the basic notions of our verification method, that ensures a valid WF process execution.

4 Firing Rules

To ensure the process consistency during runtime, we improve the modeling approach described in [8, 9] by adding some firing rules. The latter indicate under which conditions a task may fire, and what the effect of the firing on the marking is.

1. Assuming that $t_i,\ t_j \in T$, are two successive tasks in a WF, and $s \in S$ is an output data structure of t_i and an input data structure to t_j. Thus, tokens produced by t_i will be automatically consumed by t_j:

$$\forall t_i,\ t_j \in T, \exists\ s \in S \mid\ < t_i,\ t_j > \Rightarrow\ pre(t_j,\ s) = post(t_i,\ s)\ . \tag{2}$$

2. Assuming that δ is the function calculating the possible markings resulting of the firing of a task $t_i \in T$ from a marking M. So, $\forall M \in \mathbb{N}^{|S|},\ t_1,\ t_2\ldots t_k \in T$:

$$\delta(\{M_1,\ M_2\ldots M_n\},\ t_1\ t_2\ldots\ t_k) = \bigcup_{M\in\{M_1,\ M_2\ldots M_n\}} \delta(\{M\},\ t_1\ t_2\ldots\ t_k) \tag{3}$$

Where: $\delta(\{M\},\ t_1\ t_2 \dots\ t_k) = \delta(\delta(\{M\},\ t_1),\ t_2 \dots\ t_k)$
$= \delta(\delta(\delta(\{M\},\ t_1),\ t_2),\ t_3 \dots\ t_k)$
$= \dots$

The function $\delta(\{M\},\ t_i)$ is defined as follows:

$$\delta(\{M\},\ t_i) = \begin{cases} \varnothing \text{ if } M < Pre(.,\ t_i), \\ \{M\ -\ Pre(.,\ t_i) + x\} \text{ otherwise.} \end{cases} \tag{4}$$

Where $x \in [Post_{Min}(.,\ t_i),\ Post_{Max}(.,\ t_i)]$, and $\delta(\varnothing,\ t_i) = \varnothing$.

We apply (3) and (4) on the example illustrated in Fig. 1, and we calculate the possible markings resulting from the firing sequence of tasks $< t_7\ t_8\ t_{11}\ t_{12}\ t_{13}\ t_{14}\ t_{15} >$:

$\{M^{[0]}\} = \{(x_8\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0),\ (x_8\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0),\ (x_8\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0),$
$(x_8\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0),\ (0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0),\ (0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0),$
$(0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0),\ (0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0)\}$

$\delta(\{M^{[0]}\},\ t_7\ t_8\ t_{11}\ t_{12}\ t_{13}\ t_{14}\ t_{15}) = \delta(\delta(\{M^{[0]}\},\ t_7),\ t_8\ t_{11}\ t_{12}\ t_{13}\ t_{14}\ t_{15})$
$= \delta(\{\{(0\ x_8\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0),\ (0\ x_8\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0),\ (0\ x_8\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0),$
$(0\ x_8\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0)\},\ \varnothing,\ \varnothing,\ \varnothing,\ \varnothing\},\ t_8\ t_{11}\ t_{12}\ t_{13}\ t_{14}\ t_{15})$
$= \delta(\delta(\{\{(0\ x_8\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0),\ (0\ x_8\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0),\ (0\ x_8\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0),$
$(0\ x_8\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0)\},\ \varnothing\},\ t_8),\ t_{11}\ t_{12}\ t_{13}\ t_{14}\ t_{15})$
$= \delta(\{\{(0\ 0\ [0,\ x_{10}]\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0),\ (0\ 0\ [0,\ x_{10}]\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0),$
$(0\ 0\ [0,\ x_{10}]\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0),\ (0\ 0\ [0,\ x_{10}]\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0)\},\ \varnothing\},\ t_{11}\ t_{12}\ t_{13}\ t_{14}\ t_{15})$
$= \dots \quad = \{\{(0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ |D_{19}| + x_{20}),\ (0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ |D_{19}| + x_{20})\},\ \varnothing\}$

3. According to (1), any loop in a WF model will cause a blocking state. In fact, if a task t_j is waiting for a data structure s tuples, and if these tuples are produced by a task t_i, which will never be executed in certain conditions: if his input data structures are not root nodes, and if $t_j \in < t_1\ t_2 \dots t_i >$, i.e. the firing sequence leading to execute t_i, this case is identified as a deadlock. In addition, even if the input data structures of t_j are root nodes, the occured cycle may cause a livelock (i.e. a loop without progress). Thus, we require that each task $t \in T$ is fired, at most once, in a sequence of tasks starting from a marking M.

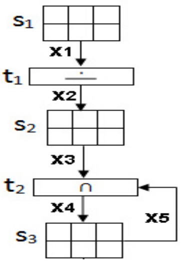
$$\forall i_1,\ i_2 \dots i_k \in \{1,\ 2 \dots |T|\},\ \delta(\{M\}, t\ t_{i_1}\ t_{i_2} \dots\ t_{i_k}\ t) = \varnothing\ . \tag{5}$$

We explain this rule through the example illustrated through Fig. 3.

4. To keep the coherency of a WF process at runtime, whatever the firing sequence, starting from a marking M, the final marking has to be the same. Formally: $\forall i_1,\ i_2 \dots i_k \in \{1,\ 2 \dots |T|\}$ and $\forall j_1,\ j_2 \dots j_k \in \{1,\ 2 \dots |T|\}$, if $\delta(\{M\},\ t_{i_1}\ t_{i_2} \dots t_{i_k}) = \varnothing$ (resp. $\delta(\{M\},\ t_{j_1}\ t_{j_2} \dots t_{j_k}) = \varnothing$) then: $\delta(\{M\},\ t_{i_1}\ t_{i_2} \dots t_{i_{k-1}}) \neq \varnothing$ (resp. $\delta(\{M\},\ t_{j_1}\ t_{j_2} \dots t_{j_{k-1}}) \neq \varnothing$) is the final marking. In such case we have to get:

$$\delta(\{M\},\ t_{i_1}\ t_{i_2} \dots t_{i_{k-1}}) = \delta(\{M\},\ t_{j_1}\ t_{j_2} \dots t_{j_{k-1}})\ . \tag{6}$$

We elucidate this rule through the example illustrated in Fig. 1.



$$
\begin{aligned}
&\{M^{[0]}\} = \left\{ \begin{pmatrix}0\\0\\0\end{pmatrix},\ \begin{pmatrix}x_1\\0\\0\end{pmatrix} \right\},\\
&\delta(\{M^{[0]}\}, t_1\ t_2\ t_2)\\
&\quad = \delta(\delta(\{M^{[0]}\}, t_1),\ t_2\ t_2)\\
&\quad = \delta\left(\left\{ \varnothing,\ \left\{ \begin{pmatrix}0\\x_2\\0\end{pmatrix} \right\} \right\},\ t_2\ t_2 \right)\\
&\quad = \delta\left(\delta\left(\left\{ \varnothing,\ \left\{ \begin{pmatrix}0\\x_2\\0\end{pmatrix} \right\} \right\},\ t_2 \right),\ t_2 \right)\\
&\quad = \delta(\varnothing,\ t_2) = \varnothing\ .
\end{aligned}
$$

Fig. 2. Process model (a) with deadlock

So, we calculate the possible markings resulting from the firing sequence of tasks < $t_7\ t_8\ t_{11}\ t_{12}\ t_{13}\ t_{14}\ t_{15}\ t\boldsymbol{.}$ >, such as $t\boldsymbol{.}$ refers to any task $\in T$ that is not in the above firing sequence:

$\delta(\{M^{[0]}\},\ t_7\ t_8\ t_{11}\ t_{12}\ t_{13}\ t_{14}\ t_{15}\ t\boldsymbol{.}) = \delta(\delta(\{M^{[0]}\},\ t_7),\ t_8\ t_{11}\ t_{12}\ t_{13}\ t_{14}\ t_{15}\ t\boldsymbol{.})$

$= \ldots$

$= \delta(\{\{(0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ |D_{19}| + x_{20}),\ (0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ |D_{19}| + x_{20})\},\ \varnothing\},\ t\boldsymbol{.}) = \varnothing$

$\Rightarrow \{\{(0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ |D_{19}| + x_{20}),\ (0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ |D_{19}| + x_{20})\},\ \varnothing\}$ is the final marking. We also calculate the possible markings resulting from the firing sequence of tasks < $t_7\ t_8\ t_{11}\ t_{13}\ t_{12}\ t_{14}\ t_{15}\ t\boldsymbol{.}$ >:

$\delta(\{M^{[0]}\},\ t_7\ t_8\ t_{11}\ t_{13}\ t_{12}\ t_{14}\ t_{15}\ t\boldsymbol{.}) = \delta(\delta(\{M^{[0]}\},\ t_7),\ t_8\ t_{11}\ t_{13}\ t_{12}\ t_{14}\ t_{15}\ t\boldsymbol{.})$

$= \ldots$

$= \delta(\{\{(0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ |D_{19}| + x_{20}),\ (0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ |D_{19}| + x_{20})\},\ \varnothing\},\ t\boldsymbol{.}) = \varnothing$

$\Rightarrow \{\{(0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ |D_{19}| + x_{20}),\ (0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ |D_{19}| + x_{20})\},\ \varnothing\}$ is the final marking. So, as t_{12} and t_{13} are two parallel tasks:

$\delta(\{M^{[0]}\}, t_7\ t_8\ t_{11}\ t_{12}\ t_{13}\ t_{14}\ t_{15}) = \delta(\{M^{[0]}\}, t_7\ t_8\ t_{11}\ t_{13}\ t_{12}\ t_{14}\ t_{15})$.

5. When the final marking has been reached, a WF process needs to be revived in order to be executed again. In other words, we have to ensure that $\forall M \in \mathbb{N}^{|S|},\ i_1,\ i_2 \ldots i_k\ \in\ \{1,\ 2 \ldots |T|\}$:

$$\delta(\{M^{[0]}\},\ t_{i_1}\ t_{i_2} \ldots t_{i_k}\ t_{i_1}) = \varnothing\ . \tag{7}$$

To do so, we extend WF by a restitution task $t_r \notin T$, i.e. t_r is not a data operation, it is a simple transition used to return from all the final states to the initial states. In such case:

$$\delta(\varnothing,\ t_r) = \{M^{[0]}\}\ . \tag{8}$$

5 Workflow Analysis

As introduced in [13], the classical soundness property grants that a process has always the possibility to terminate and all its tasks are coverable (i.e., can potentially be executed). Termination ensures that the workflow can, during its execution, neither get stuck (i.e., it is deadlock free) nor enter a loop that cannot be left (i.e., it is livelocks free), whereas coverable excludes dead tasks in the workflow. But, to ensure these criteria, the soundness property needs, to be verified, that the process has a single source place i and a single final place o.

Nevertheless, to reflect the reality of business processes, we allow a WF model to present initial and final states as needed and accordingly, it is not possible to detect this classical soundness property. So, we propose a released notion of soundness which ensures that there are no livelocks, or deadlocks, or dead tasks in a WF. In other words, we will verify the well-formedness property of a WF process. According to [14], a P-net is well-formed if it is live and bounded. We adopt this rule to WF, thus, the first step is to verify its liveness property.

5.1 Verification of the Liveness Property

We tackle this issue in [8,9], but by analyzing other process cases, we have been aware that the proposed technique is not enough to ensure the liveness property of a WF model. So, we improve it as follows:

Assuming that $\{M^{[0]}\}$ is the set of the possible initial markings, a WF model is live if and only if: $\forall t \in T,\ \exists\ t_1,\ t_2 \dots t_n \in T \mid \delta(M^{[0]},\ t_1\ t_2 \dots t_n\ t) \neq \varnothing \Rightarrow$ t is live.

To ensure the verification of this property, we proposed in [8, 9] a simple algorithm based on (9) and (10), which are defined as the following:

$$\text{Firable}(t) = \bigwedge_{\substack{i \in \{1,\ 2 \dots |S|\} \\ Pre(s_i,\ t) \neq 0}} \text{Expectable}(s_i)\ . \tag{9}$$

$$\text{Expectable}(s) = \begin{cases} \text{true if } \forall i \in \{1,\ 2 \dots |T|\},\ Post_{Max}(s,\ t_i) = 0\ . \\ \text{Firable}(t_i) \text{ if } \exists\ i \in \{1,\ 2 \dots |T|\},\ \text{ where } Post_{Max}(s,\ t_i) \neq 0\ . \end{cases} \tag{10}$$

We elucidate (9) and (10) through the example illustrated in Fig. 1.

$$\text{Firable}(t_{15}) = \bigwedge_{\substack{i \in \{8,9 \dots 19\} \\ Pre(s_i,\ t_{15}) \neq 0}} \text{Expectable}(s_i) = \text{Expectable}(s_{18}) = \text{Firable}(t_{14})$$

$$= \text{Expectable}(s_{15}) \wedge \text{Expectable}(s_{17}) = \text{Firable}(t_{12}) \wedge \text{Firable}(t_{13}) = \text{Expectable}(s_{14}) \wedge$$
$$\text{Expectable}(s_{16}) = \text{Firable}(t_{11}) \wedge \text{true} = \text{Firable}(t_{11}) = \text{Expectable}(s_{10})$$
$$= \text{Firable}(t_8) = \text{Expectable}(s_9) = \text{Firable}(t_7) = \text{Expectable}(s_8) = \text{true}$$

By using (9) and (10), we can verify that every task in a WF process is firable if its expected tokens can be provided by the evolution of the marking. However, this is not sufficient to verify its liveness proverty. In fact, if a WF model contains structural conflicts, there will be a part in the workflow that may not be executed. So, before applying (9) and (10), we have to start by verifying that the workflow does not contains structural conflicts.

Conflict Resolution: we assume that a WF model has a structural conflict, if it contains at least two tasks t_i and t_j having the same input data structure s_k, e.g., t_2 and t_5 sharing s_2 in the *Role* 1 sub-process, t_9 and t_{11} sharing s_{10}

in the *Role* 2 sub-process. To resolve such conflicts [8,9], we extend the model by adding extra tasks $T^* = \{t_{clone_1},\ t_{clone_2} \dots t_{clone_l}\}$ such as l is the number of conflict tasks, and t_{clone} is a *clone* operation formally defined as follows: whether $s_k = (C_k,\ D_k)$, $t_{clone}(s_k,\ l) = \{s_{k_1}, s_{k_2} \dots s_{k_l}\}$.

The extended model $WF_+ = (S_+,\ T_+,\ Pre_+,\ Post_+)$ such as: $S_+ = S$, $T_+ = T \cup T^*$, $Pre_+ = S \times T_+$, and $Post_+ = T_+ \times S$.

Blocking State Resolution: after extending WF, the process has to be verified to ensure that there are no deadlocks or livelocks. In fact, if we apply (9) and (10) directly on a WF_+, which contains deadlocks or livelocks, the equations will enter in an infinite loop, as the case of model (a) presented in Fig. 2:

$$\text{Firable}(t_2) = \bigwedge_{\substack{i \in \{1,\ 2,\ 3\} \\ Pre(s_i,\ t_2) \neq 0}} \text{Expectable}(s_i) = \text{Expectable}(s_2) \wedge \text{Expectable}(s_3)$$

$$= \text{Firable}(t_1) \wedge \text{Firable}(t_2) = \text{Expectable}(s_1) \wedge \text{Expectable}(s_2) \wedge \text{Expectable}(s_3)$$

$$= \text{true} \wedge \text{Firable}(t_1) \wedge \text{Firable}(t_2) = \text{Firable}(t_1) \wedge \text{Firable}(t_2) = \text{Expectable}(s_1) \wedge \text{Expectable}(s_2) \wedge \text{Expectable}(s_3) = \text{true} \wedge \text{Firable}(t_1) \wedge \text{Firable}(t_2) = \dots$$

The verification of deadlocks or livelocks is ensured by (5) defined by the firing rule 3, which prohibits the existence of loops in a WF model. If this rule is not verified, it means that the model contains deadlocks or livelocks and accordingly, it is not live.

5.2 Verification of the Boundedness Property

As we extended WF to WF_+, the boundedness property will be verified relatively to the extended model. If WF_+ is not bounded, it means that the workflow will contain at least one data structure having a number of tokens increasing infinitely with the evolution of the marking. To verify the boundedness property of a WF, we assume that if its WF_+ has no loop, it will be bounded. We prove this idea as follows: According to (2): $\forall t_i,\ t_j \in T,\ \exists s \in S \mid\ < t_i,\ t_j >$, $pre(t_j,\ s) = post(t_i,\ s)$, which ensures that the number of tokens produced by a task, in its output data structure, will be automatically consumed by the next task having, as input, the same data structure. Besides, according to (5): $\forall i \in \{1,\ 2\ \dots |T|\},\ M \in \{M_1, M_2 \dots M_n\},\ \delta(\{M\}, t\ t_{i_1}\ t_{i_2} \dots\ t_{i_k}\ t) = \varnothing$, which means that, there is no cycle in a WF model.

Consequently, $\{i \in \{1,\ 2 \dots |T|\},\ M > Pre(.,\ t_i)\} = \varnothing$, and accordingly, in any case, the marking of a data structure will never be higher then the number of tuples requested by the task consuming this data structure tuples. Thus, $|\bigcup_{\theta=0}^{+\infty} \delta(M^{[\theta]},\ t_i)| < +\infty$, which means that, the set of possible markings $\delta(\{M^{[\theta]}\},\ t_i)$ is a finite set $\forall t_i \in T$, and consequently, WF_+ is bounded.

6 Opus Framework

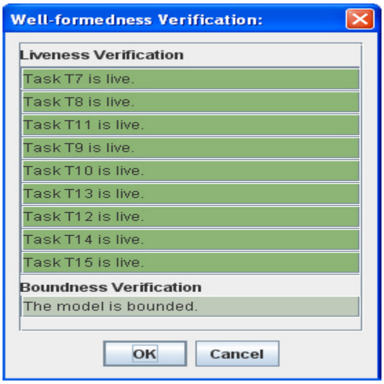
Opus Framework is implemented using Java Swing language with a set of Java library, namely, JGraph, UMLGraph, JTable... It consists of a number of components including a modeling editor, a workflow engine, and a verification module.

6.1 Opus Editor

The graphical modeling of workflow processes is ensured using *Opus editor*. The latter is equipped with a set of graphical interfaces to create profiles of roles performing the work, define data flow interactions between roles and the IS (e.g. ICRole receives the data structure *Customer_Inf* from SCRole and saves *Order_Table* and *Order_Line_Table* tokens in the IS), and finally, define the sub-process model related to each role work. It also provides to the designer a customized assistant for each operation in the process model, in order to help him to model the process structure.

6.2 Verification of the Workflow Model

Opus system is equipped with a *verification module* which ensures the analyses and the verification of the conceived workflow models, as described in Sect. 5. The verification result of the ICRole sub-process is illustrated in Fig. 3. We illustrate also the verification of model (a) (defined in Fig. 2), through Fig. 4.


Well-formedness Verification:

Liveness Verification

Task T7 is live.

Task T8 is live.

Task T11 is live.

Task T9 is live.

Task T10 is live.

Task T13 is live.

Task T12 is live.

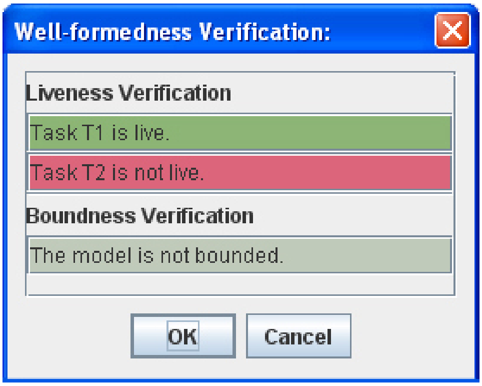
Task T14 is live.

Task T15 is live.

Boundness Verification

The model is bounded.

Fig. 3. ICRole sub-process verification (processing time 1.37 sec)


Well-formedness Verification:

Liveness Verification

Task T1 is live.

Task T2 is not live.

Boundness Verification

The model is not bounded.

Fig. 4. Model (a) verification (processing time 0.24 sec)

6.3 Opus Engine

Opus engine follows up the data flow routing, simulates the processing of tasks according to its formal definition, considering the firing rules defined in Sect. 4, and invites each role to perform its tasks according to its feasibility and urgency. Furthermore, tokens of workflow initial states may be imported from the IS. And in the same way, tokens of final states may be stored in. For this purpose, *Opus* engine is equipped with the *IS Integration Module* that provides the *Import tool* (which imports tuples from a definite IS table to a definite data structure [9]), the *ImportId tool* (which imports the identifier of the last tuple inserted in a definite IS table, instead of being entered by a role), and the *Insert*

tool (which stores a data structure tuples in a definite IS table. It is considered as an *Add_Tuples* operation such as t_{10} and t_{15} in Fig 1). We detail all the actions, performed either by the ICRole or by *Opus* engine, through Fig. 5.
A_1. *Starting the ICRole sub-process:* when the *ICRole* launches his process, the engine will present to him the data structure *Customer_Inf* received from *SCRole* and will ask him to instantiate the data structure *Product_Table.*

A_2. *Alimentation of the data structure Product_Table from the IS:* the *ICRole* imports tuples to *Product_Table* from the IS using the *Import* tool, and validates its tokens in order to execute t_7 (see Step 1).

A_3. *Seizure of the ordered quantities:* the *ICRole* seizes the ordered quantities relatively to the ordered products (OrdPs), in the resulting data structure of t_7, and validates his seizures (see Step 2).

A_4. *Select the OrdPs:* during the execution of t_8, the engine invites the *ICRole* to enter the selection property in order to select the OrdPs (see Step 3). Then, the *ICRole* saves the selection property for the next executions of t_8.

A_5. *Verify the availability of the OrdPs:* in this runtime example, $s_{10} \neq \emptyset$, so, t_{11} decides to send s_{14} to t_{12}, also t_9 decides to send s_{11} to t_{10}.

A_6. *Insert a new order:* t_{10} uses the *Insert tool* to insert a new order in the IS *Orders* Table (see Steps 5, 5.1, 5.2).

A_7. *Create the new order lines:* in parallel with t_9, t_{11} then t_{12} will be automatically executed to produce s_{15} (see Step 4-2). Then, t_{14} will be waiting for s_{17} to be executed. In this case, the *ICRole* has to launch the execution of t_{13}.

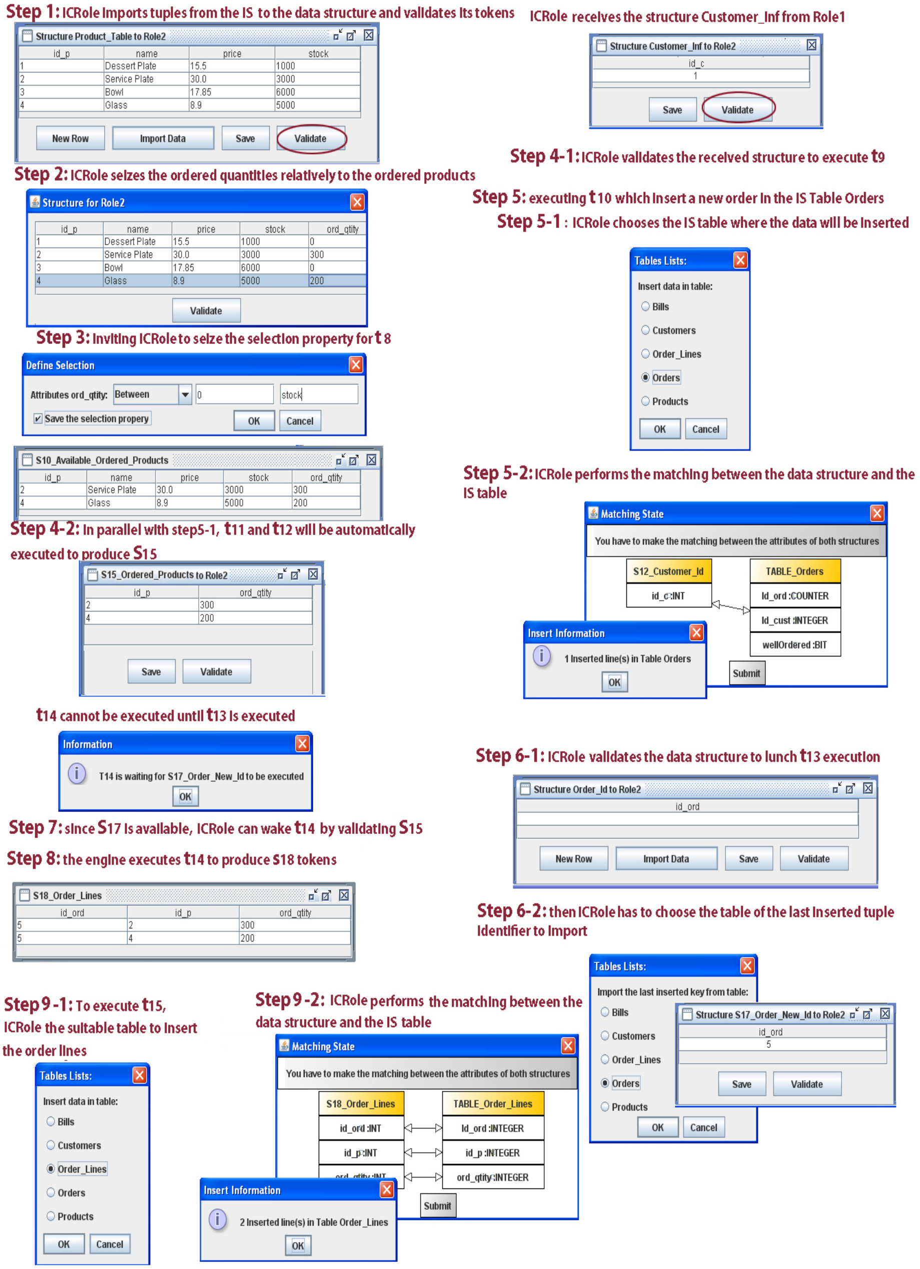
A_8. *Import the identifier of the last inserted order from the IS:* the *ICRole* launches the execution of t_{13} (see Step 6-1). The latter receives the empty data structure *S16_Order_Id* as an input, and instead of seizing the new order identifier, t_{13} will import its value using the *ImportId* tool (see Step 6-2).

A_9. *Wake a waiting task:* at this level, the *ICRole* can turn to wake t_{14} by validating s_{15} tokens, and the engine will launch its execution (see Step 7).

A_{10}. *Complete the creation of the new order lines:* the engine executes t_{14} to produce s_{18} tokens (see Step 8).

A_{11}. *Insert the new order lines:* the engine executes t_{15} and asks the *ICRole* to choose the suitable IS table for the insertion (see Step 9-1), and to perform the matching between the data structure s_{18} and the chosen table (see Step 9-2). If these two steps are well done, the engine will properly end the workflow.

We can deduce, from the execution details, the presence of four types of actions: 9.1% of actions are based on *manual tasks* (i.e. tasks that are performed only by a role without the intervention of the engine, such as A_3), 27.27% of actions are based on *automatic tasks* (i.e. tasks that are performed only by the engine without the intervention of a role, such as A_5, A_7 and A_{10}), 18.18% of actions are based on *semi-automatic tasks* (i.e. tasks that are performed by the engine under control of a role, such as A_1 and A_9), and 45.45% of actions are based on *semi-automatic tasks only in the first execution (SA-FE)* (i.e. tasks that are semi-automatic tasks only in their first executions, but during their next executions, they will migrate to be automatic tasks, such as A_2, A_4, A_6, A_8 and A_{11}).



Step 1: ICRole imports tuples from the IS to the data structure and validates its tokens

Structure Product_Table to Role2

id_p	name	price	stock
1	Dessert Plate	15.5	1000
2	Service Plate	30.0	3000
3	Bowl	17.85	6000
4	Glass	8.9	5000

ICRole receives the structure Customer_Inf from Role1

Structure Customer_Inf to Role2

id_c
1

Step 2: ICRole seizes the ordered quantities relatively to the ordered products

Structure for Role2

id_p	name	price	stock	ord_qtity
1	Dessert Plate	15.5	1000	0
2	Service Plate	30.0	3000	300
3	Bowl	17.85	6000	0
4	Glass	8.9	5000	200

Step 3: inviting ICRole to seize the selection property for t 8

Define Selection

Attributes ord_qtity: Between | 0 | stock

☑ Save the selection propery

S10_Available_Ordered_Products

id_p	name	price	stock	ord_qtity
2	Service Plate	30.0	3000	300
4	Glass	8.9	5000	200

Step 4-1: ICRole validates the received structure to execute t9

Step 4-2: In parallel with step5-1, t11 and t12 will be automatically executed to produce S15

S15_Ordered_Products to Role2

id_p	ord_qtity
2	300
4	200

Step 5: executing t 10 which insert a new order in the IS Table Orders

Step 5-1 : ICRole chooses the IS table where the data will be inserted

Tables Lists:

Insert data in table:

- ○ Bills
- ○ Customers
- ○ Order_Lines
- ● Orders
- ○ Products

Step 5-2: ICRole performs the matching between the data structure and the IS table

Matching State

You have to make the matching between the attributes of both structures

S12_Customer_Id	TABLE_Orders
id_c :INT	Id_ord :COUNTER
	Id_cust :INTEGER
	wellOrdered :BIT

Insert Information

1 Inserted line(s) in Table Orders

t14 cannot be executed until t13 is executed

Information

T14 is waiting for S17_Order_New_Id to be executed

Step 6-1: ICRole validates the data structure to lunch t13 execution

Structure Order_Id to Role2

id_ord

Step 6-2: then ICRole has to choose the table of the last inserted tuple identifier to import

Tables Lists:

Import the last inserted key from table:

- ○ Bills
- ○ Customers
- ○ Order_Lines
- ● Orders
- ○ Products

Structure S17_Order_New_Id to Role2

id_ord
5

Step 7: since S17 is available, ICRole can wake t14 by validating S15

Step 8: the engine executes t14 to produce S18 tokens

S18_Order_Lines

id_ord	id_p	ord_qtity
5	2	300
5	4	200

Step 9-1: To execute t15, ICRole the suitable table to insert the order lines

Tables Lists:

Insert data in table:

- ○ Bills
- ○ Customers
- ● Order_Lines
- ○ Orders
- ○ Products

Step 9-2: ICRole performs the matching between the data structure and the IS table

Matching State

You have to make the matching between the attributes of both structures

S18_Order_Lines	TABLE_Order_Lines
id_ord :INT	Id_ord :INTEGER
id_p :INT	id_p :INTEGER
ord_qtity :INT	ord_qtity:INTEGER

Insert Information

2 Inserted line(s) in Table Order_Lines

Fig. 5. Executing the ICRole sub-process

7 Conclusions and Future Work

According to the execution of the ICRole sub-process, manual tasks are extremely reduced. Other tasks are either purely executed by the engine, or executed by the engine under supervising of a human actor. That demonstrates the success of the modeling approach in execution issue. In fact, thanks to the detailed definition of the workflow model, *Opus* engine can interpret, automatically, the process operational functions and perform a data-driven execution based on the firing rules defined in Sect. 4. These latter ensure the consistency of data, during runtime, and grant, together with the verification method, presented in Sect. 5, the proper termination of the workflow process. However, this Framework must be completed by a module for documents generation (invoice, purchase order...): the system can manage the content but not the container.

References

1. van der Aalst, W.M.P., Hee, K.: Workflow Management: Models, Methods, and Systems. MIT Press (2004)
2. Künzle, V., Reichert, M.: Philharmonicflows: towards a framework for object-aware process management. Journal of Software Maintenance and Evolution: Research and Practice 23(4), 205–244 (2011)
3. Delzanno, G.: An overview of msr(c): A clp-based framework for the symbolic verification of parameterized concurrent systems. Electr. Notes Theor. Comput. Sci. 76, 65–82 (2002)
4. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Syst. J. 42(3), 428–445 (2003)
5. Müller, D., Reichert, M., Herbst, J.: Data-driven modeling and coordination of large process structures. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 131–149. Springer, Heidelberg (2007)
6. Lazic, R., Newcomb, T.C., Ouaknine, J., Roscoe, A.W., Worrell, J.: Nets with tokens which carry data. Fund. Informaticae 88(3), 251–274 (2008)
7. Sidorova, N., Stahl, C., Trčka, N.: Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible. Inf. Syst. 36(7), 1026–1043 (2011)
8. Haddar, N., Tmar, M., Gargouri, F.: A data-driven workflow based on structured tokens petri net. In: The Seventh International Conference on Software Engineering Advances, ICSEA 2012, pp. 154–160 (2012)
9. Haddar, N., Tmar, M., Gargouri, F.: Implementation of a data-driven workflow management system. In: IEEE 15th International Conference on Computational Science and Engineering, CSE 2012, pp. 111–118. IEEE Computer Society (2012)
10. Liu, D., Wang, J., Chan, S.C.F., Sun, J., Zhang, L.: Modeling workflow processes with colored petri nets. Comput. Ind. 49(3), 267–281 (2002)
11. Aalst, W., Weske, M., Grünbauer, D.: Case handling: a new paradigm for business process support. Data Knowl. Eng. 53(2), 129–162 (2005)
12. Künzle, V., Reichert, M.: Philharmonicflows: Research and design methodology. Technical report, University of Ulm (May 2011)
13. Aalst, W.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997)

14. van der Aalst, W.M.P.: Workflow verification: Finding control-flow errors using petri-net-based techniques. In: van der Aalst, W.M.P., Desel, J., Oberweis, A. (eds.) Business Process Management. LNCS, vol. 1806, pp. 161–183. Springer, Heidelberg (2000)

Appendix A: Data Operations Definition

Table 1. Operations definition

Operation	**Formal definition**
Inner Product: performs the combination of a data structure tuples with those of another data structure. Noted: ×	$\forall\ s_j = (C_j,\ D_j),\ s_k = (C_k,\ D_k),\ C_j = (c_{j_1},\ c_{j_2} \dots c_{j_{n_j}}),$ $C_k = (c_{k_1},\ c_{k_2} \dots c_{k_{n_k}}), s_i = s_j \times s_k = ((c_{j_1} \dots c_{j_{n_j}}, c_{k_1} \dots c_{k_{n_k}}), D_i)$ Where: $D_i = \bigcup_{\substack{l \in \{1 \dots n_j\} \\ p \in \{1 \dots n_k\}}} \{(d_{j_l}{}^{1} \dots d_{j_l}{}^{n_j}, d_{k_p}{}^{1} \dots d_{k_p}{}^{n_k})\}$ Resulted tokens number:$\lvert D_i \rvert = \lvert D_j \rvert \times \lvert D_k \rvert$
Selection: selects the tuples of a data structure that meet the desired criteria. Noted: σ	Whether P is the selection property, $\forall\ s_j = (C_j,\ D_j), s_i = \sigma_P s_j$ $\Leftrightarrow s_i = (C_j, \bigcup_{\substack{e \in D_j \\ P(e)}} \{e\})$ Resulted tokens number:$\lvert D_i \rvert \in [0,\ \lvert D_j \rvert]$
Projection: selects the values of specific attributes in a data structure. Noted: $\dot{-}$	$s_j = (C_j, D_j), \forall (b_1 \dots b_n) \in \{0,1\}^n,\ s_i = (C_i, D_i) = \dot{-}_{(b_1 \dots b_n)} s_j$ Where c_i is a selected (resp. not selected) attribute, if $b_i = 1$ (resp $b_i = 0$). $\Leftrightarrow C_i = (c_{j_{j'_1}},\ c_{j_{j'_2}} \dots c_{j_{j'_q}}),\ D_i = \{(d_{j_{1_{j'_1}}},\ d_{j_{1_{j'_2}}} \dots d_{j_{1_{j'_q}}}),$ $(d_{j_{2_{j'_1}}},\ d_{j_{2_{j'_2}}} \dots d_{j_{2_{j'_q}}}) \dots (d_{j_{m_j}{}_{j'_1}}, d_{j_{m_j}{}_{j'_2}} \dots d_{j_{m_j}{}_{j'_q}})\}$. Such as: $q = \sum_{k=1}^{n} b_k$: is the number of attributes in the structure result, $j'_k = \min_{\substack{l = \{1,2 \dots n\} \\ \sum_{p=1}^{l} b_p = k}}$ l: refers to the projection attributes indices. Resulted tokens number: $\lvert D_i \rvert \in [0,\ \lvert D_j \rvert]$
Substitution: changes the name of an attribute in a data structure. Noted: $\boxslash$	$\forall s_j = (C_j,\ D_j),\ s_i = \boxslash(c_{j_k},\ c,\ s_j) = ((c_{j_1} \dots c_{j_k -1}, c, c_{j_k +1} \dots c_{j_n}),\ D_j)$ Resulted tokens number: $\lvert D_i \rvert = \lvert D_j \rvert$
Permutation: allows to permute two columns in a data structure. Noted: $\circlearrowright$	$\forall s_i = (C_i,\ D_i), s_j = \circlearrowright(s_i,\ k,\ l)$, such as: $k,\ l \in \{1,2 \dots n\},\ k < l,$ $C_j = (c_{i_1} \dots c_{i_{k-1}}, c_{i_l}, c_{i_{k+1}} \dots c_{i_{l-1}}, c_{i_k}, c_{i_{l+1}} \dots c_{i_n})$ $D_j = \{(d_{1\,i_1} \dots d_{1\,i_{k-1}}, d_{1\,i_l}, d_{1\,i_{k+1}} \dots d_{1\,i_{l-1}}, d_{1\,i_k}, d_{1\,i_{l+1}} \dots d_{1\,i_n}),$ $(d_{m\,i_1} \dots d_{m\,i_{k-1}}, d_{m\,i_l}, d_{m\,i_{k+1}} \dots d_{m\,i_{l-1}}, d_{m\,i_k}, d_{m\,i_{l+1}} \dots d_{m\,i_n})\}.$ Resulted tokens number:$\lvert D_i \rvert = \lvert D_j \rvert$
Extension: Extends a structure scheme by adding an attribute c =(n, t) and applying a function f. Noted: $\underset{\cdot}{-}$	$\forall s_j = (C_j,\ D_j),\ s_i = \underset{\cdot}{-}(s_j,\ c,\ f)$, such as: $C_i = ((c_{j_1}, c_{j_2} \dots c_{j_n}, c), D_i = \{(d_{j_{1_1}}, d_{j_{1_2}} \dots d_{j_{1_n}},\ f(d_{j_{1_1}}, d_{j_{1_2}} \dots d_{j_{1_n}},$ $D_j)) \dots (d_{j_{m_1}}, d_{j_{m_2}} \dots d_{j_{m_n}}, f(d_{j_{m_1}}, d_{j_{m_2}} \dots d_{j_{m_n}},\ D_j))\})$ Resulted tokens number:$\lvert D_i \rvert = \lvert D_j \rvert$
Add_Tuples: inserts the tuples of a data structure in another one. Noted: +	$\forall s_j = (C_j,\ D_j), ,\ s_k = (C_k,\ D_k),$ $D_j = \{(d_{j_{1_1}},\ d_{j_{1_2}} \dots d_{j_{1_n}}), (d_{j_{2_1}},\ d_{j_{2_2}} \dots d_{j_{2_n}}) \dots (d_{j_{m_1}},\ d_{j_{m_2}} \dots d_{j_{m_n}})\}$ $D_k = \{(d_{k\,1_1},\ d_{k\,1_2} \dots d_{k\,1_h}),\ (d_{k\,2_1},\ d_{k\,2_2} \dots d_{k\,2_h}) \dots (d_{k\,l_1}, d_{k\,2_2} \dots d_{k\,l_h})\},$ $b_1,\ b_2 \dots b_h) \in \{1,\ 2 \dots n\}$: refers to the positions of the added values in the resulting data structure. $s_j = +(s_k, (b_1,\ b_2 \dots b_h))$ if $h < n$ then: $s_j = ((c_1,\ c_2 \dots c_n), \{(d_{j_{1_1}},\ d_{j_{1_2}} \dots d_{j_{1_n}}),\ (d_{j_{2_1}},\ d_{j_{2_2}} \dots d_{j_{2_n}})$ $\dots (d_{j_{m_1}},\ d_{j_{m_2}} \dots d_{j_{m_n}}),\ (d_{k_{1_{b_1}}},\ d_{k_{1_{b_2}}} \dots d_{k_{1_{b_h}}} \dots d_{k_{1_n}}),$ $(d_{k_{2_{b_1}}},\ d_{k_{2_{b_2}}} \dots d_{k_{2_{b_h}}} \dots d_{k_{2_n}}) \dots (d_{k_{l_{b_1}}}, d_{k_{l_{b_2}}} \dots d_{k_{l_{b_h}}} \dots d_{k_{l_n}})\})$ if $h = n$ then: $s_j = ((c_1,\ c_2 \dots c_n), \{(d_{j_{1_1}},\ d_{j_{1_2}} \dots d_{j_{1_n}}), (d_{j_{2_1}},\ d_{j_{2_2}} \dots d_{j_{2_n}})$ $\dots (d_{j_{m_1}},\ d_{j_{m_2}} \dots d_{j_{m_n}}),\ (d_{k_{1_{b_1}}},\ d_{k_{1_{b_2}}} \dots d_{k_{1_n}}),$ $(d_{k_{2_{b_1}}},\ d_{k_{2_{b_2}}} \dots d_{k_{2_n}}) \dots (d_{k_{l_{b_1}}},\ d_{k_{l_{b_2}}} \dots d_{k_{l_n}})\})$. Resulted tokens number:$\lvert D_j \rvert = \lvert D_j \rvert + \lvert D_k \rvert$
Control 1: Decides to continue or not the information flow routing, according to condition1. Noted: ±	Condition 1: if s_i is the data structure expected by the next task, and s_j is the controlled data structure, then: $s_i = s_k \pm s_j = \begin{cases} s_k\ if\ s_j = \varnothing , \\ \varnothing\ otherwise . \end{cases}$ Resulted tokens number: $\lvert D_i \rvert \in [0,\ \lvert D_k \rvert]$
Control 2: Decides to continue or not the information flow routing, according to condition 2. Noted: ∓	Condition 2: if s_i is the data structure expected by the next task, and s_j is the controlled data structure, then: $s_i = s_k \mp s_j = \begin{cases} s_k\ if\ s_j \neq \varnothing , \\ \varnothing\ otherwise . \end{cases}$ Resulted tokens number: $\lvert D_i \rvert \in [0,\ \lvert D_k \rvert]$